The feasibility of using standard Z notation in the design of complex systems

Thesis

Systems Architecture Group

Electronics Systems Engineering

Faculty of Technology

The Open University

The Feasibility of using Standard Z Notation in the Design of Complex Systems

Volume 1 of 2

by

David John Reed

BSc BA MSc MIEE CEng

4 January 1993

Thesis submitted for the Degree of Doctor of Philosophy

Preamble

This thesis is based on collections of Z schemas that describe three systems. The thesis is lengthy because all the schemas have been included in the main text; this has eliminated cumbersome cross references to appendices and other documents that would otherwise had been necessary. The schemas were also type set using the computer tool CADiZ. Including all the Z schemas in each chapter facilitated referencing and expansion by CADiZ.

EX12

The Open University
Higher Degrees Office

HIGHER DEGREES OFFICE

LIBRARY AUTHORISATION FORM

Please return this form to the Higher Degrees Office with the bound library copies of your thesis. All students should complete Part 1. Part 2 applies only to PhD students.

Student: DAVID REED PI: L0022035

Degree: PhD

Thesis title: The Feasibility of using Standard Z Notation in the Design of Complex Systems

Part 1 Open University Library Authorisation (to be completed by all students)

I confirm that I am willing for my thesis to be made available to readers by the Open University Library, and that it may be photocopied, subject to the discretion of the Librarian.

Signed: David Reed Date: 25/1/94

Part 2 British Library Authorisation (to be completed by PhD students only)

If you want a copy of your PhD thesis to be held by the British Library, you must sign a British Doctoral Thesis Agreement Form. You should return it to the Higher Degrees Office with this form and your bound thesis. *You are also required to supply a third, unbound copy of your thesis.* The British Library will use this to make their microfilm copy; it will not be returned. Information on the presentation of the thesis is given in the Agreement Form.

If your thesis is part of a collaborative group project, you will need to obtain the signatures of others involved for the Agreement Form.

The University has agreed that the lodging of your thesis with the British Library should be voluntary. Please tick either (a) or (b) below to indicate your intentions.

(a) ☑ I am willing for the Open University to supply the British Library with a copy of my thesis. A signed Agreement Form and 3 copies of my thesis are attached (two bound as specified in Section 9.4 of the Research Degree Handbook and the third unbound).

(b) ☐ I do not wish the Open University to supply a copy of my thesis to the British Library.

Signed: David Reed Date: 25/1/94

Abstract

Formal design methods are becoming increasingly recognised as being useful for specifying complex systems. Incorporating formal methods in the early stages of a design process introduces the possibility of using mathematical techniques, hence improving the effectiveness of a design process.

The Z notation has been applied mainly to specifying software, although it has also been used for specifying hardware and general systems. The Z notation fulfils two functions in this thesis. The first function is as a notation for representing specifications of complex systems, and the second function is as a notation for representing implementations of the same complex systems. The suitability of the Z notation for these functions is investigated in three studies. Both the specifications and implementations are represented as unified collections of schemas that describe the behaviour in response to each set of input conditions.

In each of the studies, both the specifications and implementations of the complex system take place at an early stage in a design process. Throughout this thesis non rigorous proof sketches prove that the implementations meet the requirements of the specifications.

Acknowledgements

I wish to express my gratitude to the Open University for supporting this research. In particular I would like to thank my supervisor Professor John Monk for all his patience and support.

Contents

Volume 1 of 2

Preamble i

Abstract ii

Acknowledgements iii

Contents iv

List of Figures xii

List of Tables xv

1 Introduction 1

1.1 Background 1

1.1.1 Formal Methods 2

1.1.2 Advantages of Formal Methods 3

1.1.3 Limitations of Formal Methods 3

1.1.4 Proof Sketches 4

1.1.5 Design Processes 5

1.2 Research Summary 7

1.2.1 Premises 7

1.2.2 Notation 7

1.2.3 The Studies 8

1.2.4 Similarities between the Studies 9

1.2.5 Differences between the Specifications of the Studies 9

1.2.6 Differences between the Proof Sketches in the Studies 9

1.3 Organization of the Thesis 10

2 The Z Notation 11

2.1 Introduction 11

2.2 Related Applications of the Z Notation 13

2.3 Unified Descriptions in the Z Notation 14

2.4 CADiZ 17

2.5 Summary 19

3 An Implementation of a Communications Network 20

3.1 Specification and Implementation 22

3.1.1 Initial Description 22

3.1.2 Informal Statement of Properties used in this Study 23

3.1.3 Informal Description of the Z Implementation 23

3.2 Data Types and Invariant 25

3.2.1 Given Sets 25

3.2.2 Free Data Types 26

3.2.3 Network State Schema 28

3.3 Network Operations 32

3.3.1 The Join Operation 32

3.3.2 The Leave Operation 46

3.3.3 The Call Operation 49

3.3.4 The Clear Operation 55

3.3.5 The Send Operation 59

3.3.6 The Receive Operation 63

3.3.7 Network Implementation at this Design Stage 69

3.4 Formal Statement of the Safety Properties 69

3.5 Additional Properties as a Consequence of the Implementation 76

3.6 Formal Specification of Liveness Properties 80

3.7 Discussion 90

3.7.1 Verification Conditions 90

3.7.2 Isolated Operation Specification 92

3.7.3 Disjunction of Schemas 93

3.7.4 Style of Writing Schemas 95

3.7.5 Concurrent Activities 96

3.8 Summary 97

3.8.1 Complete Description of an Implementation 97

3.8.2 Proof Obligations 97

3.8.3 Preconditions 98

3.8.4 Postconditions 99

3.8.5 Properties 99

3.8.6 Liveness 99

4 Replicated Database Systems 101

4.1 Introduction 101

4.2 Serializability Theory 104

4.2.1 Serialization Graphs 104

4.2.2 Serialization Graphs for Replicated Databases 107

4.2.3 Replicated Data Serialization Graphs 109

4.3 Concurrency Control Techniques in Replicated Database Systems 110

4.3.1 Two Phase Locking 111

4.3.2 Quorum Consensus 111

5 Serializability Constraint on Replicated Database Systems 113

5.1 One Copy Serializable Property 116

5.1.1 Data Definitions 116

5.1.2 Conflicting Transactions 119

5.1.3 One Copy Serialization Property 126

5.1.4 One Copy Serialization Histories 130

5.1.5 Initial State 132

5.2 Implementation of a Replicated Database System 132

5.3 Site Operations 135

5.3.1 Invariants for Site Schemas 136

5.3.2 Requests for Physical Operations 140

5.3.3 Site Response to Requests to Perform Operations 143

5.3.4 Change of Site State in Response to Execution of Operations . . 159

5.4 Management Operations 172

5.4.1 Invariants for Management Schemas 174

5.4.2 Response to Logical Operation Requests 180

5.4.3 Execution of Logical Operations 188

5.4.4 Progress of Site Requests 206

5.5 Complete Implementation 208

5.6 Proof Sketches of the Serializability of the Implementation 209

5.6.1 Proof Sketch of One Copy Serializability 210

5.6.2 Proof Sketch of the Behaviour meeting the Specification Property 212

5.7 Summary 218

Volume 2 of 2

6 Graceful Degradation of Queues 220

6.1 Introduction 221

6.2 Priority Queue 224

6.2.1 Specification of a One Copy Priority Queue 224

6.2.2 Implementation of a Replicated Priority Queue 232

6.2.3 Verification using Refinement Conditions of a Priority Queue . . . 245

6.2.4 Verification using a Proof Sketch of Equivalent Behaviour of a Priority Queue 257

6.3 Multiple Priority Queue 259

6.3.1 Specification of a One Copy Multiple Priority Queue 260

6.3.2 Implementation of a Replicated Multiple Priority Queue 267

6.3.3 Verification using Refinement Conditions of a Multiple Priority Queue 270

6.3.4 Verification using a Proof Sketch of Equivalent Behaviour of a Multiple Priority Queue 277

6.4 Out of Order Priority Queue 280

6.4.1 Specification of a One Copy Out of Order Priority Queue 280

6.4.2 Implementation of a Replicated Out of Order Priority Queue . . . 284

6.4.3 Verification using Refinement Conditions of an Out of Order Queue 286

6.4.4 Verification using a Proof Sketch of Equivalent Behaviour of an Out of Order Priority Queue 294

6.5 Degenerate Priority Queue 295

6.5.1 Specification of a One Copy Degenerate Priority Queue 295

6.5.2 Implementation of a Replicated Degenerate Priority Queue 300

6.5.3 Verification using Refinement Conditions a Degenerate Priority Queue 301

6.5.4 Verification using a Proof Sketch of a Degenerate Priority Queue 307

6.6 Lattice Structure of Behaviours 308

6.7 Summary 310

7 Conclusions 314

7.1 Complete Descriptions 314

7.2 Proof Sketches 315

7.2.1 Success of Proof Sketches 316

7.2.2 Proof Framework 317

7.2.3 Difficulties with Proof Sketches 318

7.3 Future Work 319

Glossary 321

Bibliography 324

Appendices

A Definition of Terms 336

B Examples for Chapter 3 356

B.1 CADiZ Output Examples for Chapter 3 356

B.1.1 Expansion of the Schema Pre_Join_Sch 356

B.1.2 Expansion of the Schema Simplified_3_1 357

B.1.3 Expansion of the Schema Pre_Error_Join 357

B.1.4 Expansion of the Schema Simplified_3_2 358

B.1.5 Expansion of the Schema Simplified_3_3 358

B.2 Verification of Preconditions in Schema Simplified_3_3 359

B.3 Next Stage of Design Process 363

B.3.1 Initialization Condition 369

B.3.2 Applicability Condition 370

B.3.3 Correctness Condition 370

B.4 Concurrent Activities 371

C Examples for Chapter 5 378

C.1 Example of Dead Locking History 378

C.2 Example of Transactions 378

C.3 Animation of Specification 383

C.4 Animation of Implementation 387

C.5 CADiZ Expansion of Simplified_5_3 393

C.6 Verification of Preconditions in Schema Simplified_5_3 395

C.7 Concurrent Activities 398

C.8 Disjunction and Conjunction of Schemas 407

C.9 Alternative Specification 408

D Examples for Chapter 6 418

D.1 Example of a Priority Queue 418

D.1.1 One Copy Queue 418

D.1.2 Replicated Copy Queue 419

D.2 Example of a Multiple Priority Queue 421

D.2.1 One Copy Queue 421

D.2.2 Replicated Copy Queue 423

D.3 Example of an Out of Order Priority Queue 425

D.3.1 One Copy Queue 425

D.3.2 Replicated Copy Queue 427

D.4 Example of a Degenerate Priority Queue 431

D.4.1 One Copy Queue 431

D.4.2 Replicated Copy Queue 433

D.5 CADiZ Expansion of Simplified_6_2 436

D.6 Verification of Preconditions in Simplified_6_2 439

E Syntax of the Z Notation 444

Figures

1.1 Design Process 6

2.1 Isolated Operation Schemas 15

2.2 Combination of Isolated Schemas 15

2.3 Unified Operation Schemas 16

2.4 Combination of Unified Schemas 17

2.5 Internal Structure of CADiZ 18

3.1 Overview of the Network Implementation 31

3.2 Interactions between the Join Operation Schema and the State Schema 42

3.3 Interactions between the Leave Operation schema and the State Schema 48

3.4 Interactions between the Call Operation Schema and the State Schema 53

3.5 Interactions between the Clear Operation Schema and the State Schema 57

3.6 Interactions between the Send Operation Schema and the State Schema 61

3.7 Interactions between the Receive Operation Schema and the State Schema 67

4.1 Serialization Graph for H1 106

4.2 Serialization Graph for H2 107

4.3 Serialization Graph for H3 108

4.4 Replicated Data Serialization Graph for H3 110

5.1 Construction of the One Copy Serialization Property 131

5.2 Interactions between the Management and the Site Schemas 134

5.3 Interactions between the Site State Schema and the Site Operation Schemas 136

5.4 Schema for Site Requests 140

5.5 Schemas for Site Response 143

5.6 Schemas for Site Execution 159

5.7 Main Interactions between Management Schemas 173

5.8 Mapping of the Function Site_Data 178

5.9 Management Response to Requests 180

5.10 Schemas for Management Execution 188

5.11 Schema for Site Progress 206

5.12 Inclusion Relationship for One Copy Serialization Property 213

6.1 Interactions in the Specification of a Priority Queue 224

6.2 Interactions in the Implementation of a Priority Queue 233

6.3 Schema Abstract_Queue Mappings 247

6.4 Data Refinement for Priority Queues 251

6.5 Interactions in the Specification of a Multiple Priority Queue 260

6.6 Interactions in the Implementation of a Multiple Priority Queue 268

6.7 Data Refinement for Multiple Priority Queues 273

6.8 Interactions in the Specification of an Out of Order Priority Queue 280

6.9 Interactions in the Implementation of an Out of Order Priority Queue . . 284

6.10 Data Refinement for Out of Order Priority Queues 290

6.11 Interactions in the Specification of a Degenerate Priority Queue 296

6.12 Interactions in the Implementation of a Degenerate Priority Queue 300

6.13 Data Refinement for Degenerate Priority Queues 304

6.14 Lattice Structure for Priority Queues 308

A.1 Refinement Activity 337

B.1 Interactions of the Property Oriented Implementation of the Join Operation 376

C.1 Relationship between Transactions 379

C.2 Partially Ordered History 380

C.3 Serialization Graph 381

C.4 Replicated Data Serialization Graph 382

C.5 Replicated Data Serialization Graphs 385

C.6 Interactions of the Property Oriented Implementation of the Site Request Operations 402

C.7 Interactions of the Property Oriented Implementation of the Site Read Operation 405

Tables

3.1 Summary of Design Stage in Chapter 3 22

5.1 Summary of Design Stage in Chapter 5 115

6.1 Summary of Schemas in Chapter 6 223

6.2 Analysis of Queue Implementations 311

Chapter 1
Introduction

Formal design methods are useful for specifying complex systems. They can be automated and this helps in coping with complex system designs. Incorporating formal methods during the early stages in a design process makes these stages amenable to mathematical analysis which can detect errors early on. It is particularly important to avoid errors in the early stages of a design process because of the increasing cost of re-work when errors are identified in later design stages.

This thesis investigates the use of the Z notation and proof sketches as part of a formal approach to the design of complex systems. The investigation takes the form of three studies. Each study represents an early design stage of a complex system before the system is partitioned into functional subsystems. Standard Z notation expresses complete descriptions of both a specification and an implementation. Each of the three studies exhibits a different aspect of the relationship between a specification and an implementation. In all the three studies, proof sketches are used to verify an implementation correct with respect to a specification.

1.1 Background

Computer hardware description languages (sometimes referred to as design languages) have been developed to specify and design the hardware components of computers, that is digital

electronic circuits [Dud83, Milne88]. Most of the languages developed have been for describing the functional and structural properties of hardware, leaving the description of other properties of hardware to ad hoc methods.

Not all computer hardware description languages use formal languages. The lack of formality means that the descriptions cannot be verified formally. At present there are a number of formal languages used in hardware design methods, such as CIRCAL [Milne86] and HOL [Gord85], plus a number of semi formal ones, such as VHDL [Aylor86, Marsh86], MoDL [Smit87] and ELLA [Mori85].

1.1.1 Formal Methods

A formal method for a design process is a set of well defined procedures and practices with their associated notations. Formal methods, in general, encompass other practices in addition to the notations applied in the descriptions of implementations. The additional practices include such topics as configuration control and design team organisation. The term *formal method* has caused some confusion because of the problem of defining the term *method* satisfactorily. Nicholls suggested that instead of the term *method* the three terms *process, stage* and *technique* should be used [Nich92]. A *process* is an approach taken in the design of a system. A *stage* is an identifiable phase in the process of design. A *technique* is a set of rules applied within a stage.

The important characteristic of a formal method, in the context of this thesis, is that it is based on a mathematically formal system [Gibb88]. The mathematical basis of formal techniques enables theorems about implementations to be proposed and verified. Camurati and Prinetto stated that implementations are verified with respect to specifications by two broad approaches [Cam88]:

1 Specify the required behaviour of the system in a formal language and then prove that any subsequent implementation is mathematically implied by the specification.

2 Specify properties possessed by all correct implementations in a formal language and then prove rigorously that a proposed implementation exhibits the required properties.

Both approaches have been investigated in the three studies of the research reported in this thesis.

1.1.2 Advantages of Formal Methods

It is a widely held view in the academic community that using a formal language reduces the chances of inconsistencies, unintended ambiguity and incompleteness.

One of the major advantages of formal methods is that, if both the specification and implementations are expressed in formal notations, then it is theoretically possible to verify that the implementations are correct with respect to their specifications [Wing90]. The correctness is based on mathematical analysis that can cover all possible conditions. Other possibilities such as testing or simulating can only demonstrate correctness under a subset of the possible conditions. In practice, strict mathematical verification may not be feasible because of the magnitude of the task, so informal techniques have to be applied.

A formal design process is one in which all implementations are verified with respect to specifications that are derived rigorously from the initial specification. This reduces some of the uncertainty and vagueness that occurs with informal specifications. Morgan expressed the opinion that one of the benefits of a formal specification is that it forces a decision to be made on every issue within the context of the specification: 'mathematical notation cannot be vague' [Morg83]. The implication is that the mathematical basis of formal languages encourages clarity of thought.

Informal methods using mathematical analysis based on heuristic techniques can support a design activity, but because of the lack of mathematical rigour, less confidence is likely to be attached to the results.

1.1.3 Limitations of Formal Methods

Implementations can only be proved correct with respect to a specification. The imprecise nature of how ideas are formulated in the human mind means that there is no formal way of ensuring that the specification is a correct representation of the required system [Cohn89]. Implementations can be verified completely with respect to their specifications, but still be incorrect because of the mismatch between the specifications and what is actually required.

Wing described the intrinsic bounds that are placed on formal techniques due to the informal mapping between the 'ideal' and 'real' worlds [Wing90]. In addition, she states that assumptions can be made about the systems environment which are not valid under all cir-

cumstances. This point is stressed by Pyle who warned that formality by itself does not ensure reliability in software engineering [Pyle89]. Pyle maintained that human skills of judgement, awareness and forethought are also necessary to design reliable systems.

Any description of a system represents a limited view, or an abstraction, of the system and features of the requirements may be omitted because of the inconvenience of expressing them in the chosen formal language. Once the abstraction process has been completed, it is difficult to see what is missed out. This leads designers to focus on the representation of the problem and ignore the actual problem.

There is a theoretical limit to proving valid statements expressed in a formal language. This limit is embodied in Gödel's theorem which states that it may not be possible to prove certain valid statements correct within a formal system or language. In practice the consequences of Gödel's theorem may not significantly limit the application of formal techniques. The inability to prove some statements can be overcome by accepting a proof which is dependent on assumptions about the behaviour of the system. The assumptions are in the form of additional axioms, thereby placing restrictions on the validity of the proof. Alternatively, it may be possible to change the statements to a form that can be proved correct.

1.1.4 Proof Sketches

The difficulty of analysing complex systems because of the combinatorial explosion of the number of possible behaviours leads to *proof sketches*. Proof sketches are partial proofs. They indicate how completely formal proofs can be constructed, but do not include all the links required in formal chains of reasoning. Completely formal proofs are called *proof demonstrations*. Each statement in a proof demonstration is either an axiom or a consequence of applying the rules of inference to previous statements.

Although formal notations make verification by formal proofs possible, for realistic problems proof demonstrations are not feasible because of their complexity. This is one of the reasons why some practitioners of formal methods advocate a liberal approach to verification, that is, where formal proofs are not considered essential [Nich92]. The application of a liberal approach in this thesis is to use proof sketches and not proof demonstrations.

1.1.5 Design Processes

The term *process* describes the approach taken to design, and not the actual events that occur in the act of designing. The main features of a design process presented here are that each stage in a design process has a specification and each stage results in a representation of an implementation[1] that meets the given specification. To ensure the continuity of the design process, the implementation at one design stage becomes the specification for the next stage [Milne86]. The goal in following a design process is to create an implementation in a form suitable for directing the construction of a system that meets the initial specification. In creating an implementation several intermediate forms are likely to be developed. The intermediate forms are in terms of components or features that cannot be realised directly. The final representation is in terms of components that are readily available. These components may be physical objects or instructions in an established programming language.

In the context of a design process, a specification is a statement of the required behaviour of an implementation. Specifications can be written by stating the properties that must be possessed by an implementation. The term *property* can signify different characteristics required of a system. For example, properties can specify:

1 Functional behaviour: for example, a database system is to read the latest value written to a data object when requested, or a priority queue is to return the data object with the highest priority when data is requested.

2 Performance properties: for example, the response time required for every operation, or the minimum number of simultaneous operations that must be possible.

3 Reliability: for example, the maximum number of acceptable errors, or the availability of the system.

4 Qualitative features: such as, ease of use, attractive appearance, or maintainability.

The properties considered in this thesis specify predominantly the functional behaviour of a system.

---

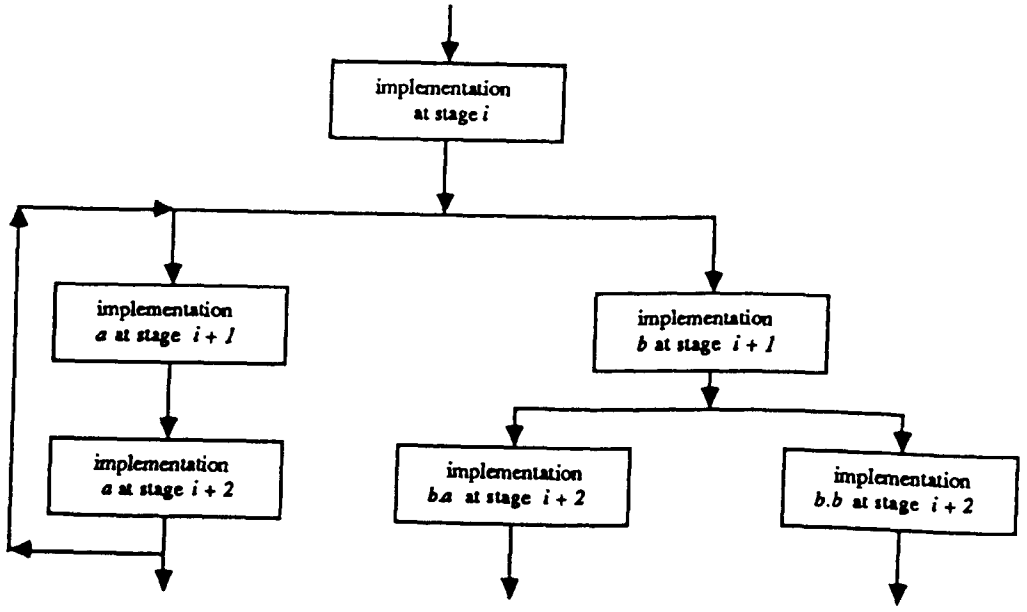
[1] *The term* 'representation of an implementation' *is abbreviated to* 'implementation' *throughout this thesis.*

The properties given in the initial specification are relevant throughout the design process and more properties may be added during the design process. Additional properties that are made explicit can have their relevance questioned at later stages of design. Properties introduced at one design stage, if relevant, are applied in all subsequent stages.

At any design stage it may be possible to verify that all the properties are satisfied by the implementation constructed at that stage. However, this is not always the case and the abstractions may not be sufficiently detailed to verify that the proposed implementation will have certain properties.

Normally, design is not a linear sequence of stages culminating in a final representation. Usually it involves exploring the ways in which various implementations will satisfy the properties and re-evaluating previous design decisions. Each design stage includes decisions that affect subsequent design stages. These decisions are taken after weighing up the advantages and disadvantages of each possible implementation, then selecting the most suitable based on value judgements. Different proposed implementations may be carried forward in parallel through a number of stages before an option is eliminated. Figure 1.1 gives a diagrammatic view of the hierarchy of design stages.

Figure 1.1 Design Process



The design process illustrated above is a simplified interpretation of the plausibility driven approach described in detail elsewhere [Agü87, Dasg87].

1.2 Research Summary

1.2.1 Premises

The two premises of the thesis that are being tested in the research are:

1 that the standard Z notation can express both the properties required of complex systems and complete descriptions of implementations of those systems

2 that using a formal notation without invoking the rigour of mathematical proof demonstrations has advantages over informal notations in terms of reducing the ambiguity and increasing the conciseness of descriptions.

The two premises are investigated by studies of the application of the standard Z notation to specify and implement three complex systems at a single design stage. The studies are used to provide a framework for discussing the problems of using the Z notation and proof sketches for practical systems.

1.2.2 Notation

The Z notation is employed in this thesis because it is widely used in industry and academia. In addition, computer tools are now available which increase the ease of using the Z notation.

The Z notation is flexible and it is tempting to make minor amendments to the notation to meet the needs of the current problems. Such temptations have been resisted and no deviations from the standard Z notation [ZipBS91] have been made in the three studies. Standard Z notation has been enforced by the CADiZ computer tool [CADiZ91, Jord91] which checks the syntax of the schemas and type sets the descriptions expressed in the Z notation.

The primary published applications of the Z notation have been to specify aspects of the behaviour required of systems in terms of states and the operations that change the state. In those applications, the specification of a system contains isolated schemas that refer to distinct operations and parts of the composite state. In contrast, the Z notation is applied in this thesis as a design language for describing the behaviour of complete systems as integrated

entities. The integrated nature of the models provides information about the flow of control, signals, inputs and outputs which is necessary for the interpretation of the formal text. An example of the differences in the approaches is given in Section 2.3.

1.2.3 The Studies

The first study is contained in Chapter 3 and is a description in the Z notation of an implementation of a communications network. Schema equations indicate the structure of the model of the system and different interpretations of the equations are discussed. This chapter also includes an example of using schema equations to describe liveness properties of a system. The first study includes many examples of simple theorems about properties of a communications network and examples of proof sketches for verifying these theorems.

The other two studies are about the concurrency control of transactions executed on replicated database systems. Replicated database systems are inherently complex systems and have properties that are difficult to specify formally. They provide a challenging bench mark for the techniques being proposed.

Chapter 5 contains a specification of the one copy serializability property and an implementation of a replicated database system that is controlled by a two phase locking algorithm [Bern87]. The property schemas and implementation schemas are developed separately. The verification condition is that the implementation (expressed in the Z notation) implies the one copy serializability property (also expressed in the Z notation). The purpose of this study is to illustrate the effectiveness and limitations of the standard Z notation applied to a complex system.

Chapter 6 uses a different way of specifying with the Z notation in the investigation of graceful degradation characteristics of a quorum consensus algorithm in the context of priority queues [Herl91]. In this chapter, two model oriented descriptions are created; one representing the specification and the other representing the implementation. The proof obligation required in this chapter is that the two models are equivalent in terms of the types of possible behaviours. This chapter provides important illustrations of the data refinement rules for the Z notation and some of the difficulties that can arise when strict equivalence between a specification and an implementation is not required.

Sections 1.2.4 to 1.2.6 summarise the main similarities and differences between the three studies.

1.2.4 Similarities between the Studies

The principal similarities between the three studies are:

1 Each implementation is represented in the Z notation as a set of schemas. To provide complete descriptions, the operations are explicitly identified with values for inputs in the schemas.

2 Proof sketches discharge proof obligations that are about the means of implementation.

3 Proof sketches verify the implementations.

1.2.5 Differences between the Specifications of the Studies

The specification for each study differ in the following ways.

1 Communications Network

The specification is in the form of a set of properties. The properties are initially stated informally and later formalised in the context of the implementation.

2 Replicated Database System

The specification is in the form of a single property based on existing mathematical theory and re-expressed in the Z notation as a set of Z schemas.

3 Distributed Queues

Four queues are specified and implemented. The specifications are given as models of the required behaviour. The models are expressed in the Z notation.

1.2.6 Differences between the Proof Sketches in the Studies

The use of proof sketches in the verifications differ in the following ways.

1 Communications Network

Each property is expressed in predicate logic as a consequence of a theorem. The verification takes the form of proof sketches confirming the theorems.

2 Replicated Database System

Two approaches are taken with the verification of the implementation. The first approach uses a proof sketch to verify that the implementation satisfies the Z notation expression of a mathematical property. The second approach uses a proof sketch to verify that the implementation meets an informal understanding of the required property.

3 Distributed Queues

The verification takes the form of showing the equivalence between a model of a specification and a model of an implementation. Two approaches are taken with the verification. The first approach is based on using proof sketches to verify the data refinement conditions for the Z notation. The second approach is based on proof sketches to demonstrate the similarities of the behaviours inherent in the specification and implementation of each type of queue.

1.3 Organisation of the Thesis

The whole description in the Z notation of each study is included in each associated chapter. This has the effect of making the chapters containing the studies appear quite long. The whole descriptions are retained to indicate the size of the representations required at a high level of abstraction. The schemas are integrated into the text to avoid excessive cross referencing. Diagram are used throughout the text to illustrate the interactions between schemas.

The three studies are contained in Chapters 3, 5 and 6.

Chapter 2 contains a review of the applications of the Z notation and an explanation of how the Z notation is used in this thesis.

Chapter 4 introduces the topic of replicated database systems in preparation for an analysis of these systems.

Chapter 7 presents the main conclusions of the research.

The terms used in this thesis are defined informally in Appendix A as an aid to reading subsequent chapters. In addition, a glossary of symbols is provided in a separate chapter.

Chapter 2
The Z Notation

This chapter assumes some knowledge of the Z notation. The level of detail contained in the articles by Spivey [Spiv89B] and Woodcock [Wood89B] is sufficient to understand the descriptions and applications of the Z notation reviewed in this chapter.

2.1 Introduction

The Z notation was initially used in 1978 at the Programming Research Group in the Oxford University Computing Laboratory. The device of a schema to structure specifications was included in the Z notation in 1982. The Z notation has been applied mainly to specifying software, although it has been used for specifying hardware and general systems [Hayes87]. At the time of writing, the Z notation is being standardised in anticipation of being accepted internationally.

The Z notation is a language and a style for expressing formal specifications of computing systems [Spiv88, Spiv89A, Spiv89B]. It is based on a typed set theory and the concept of a schema is one of its key features. A schema consists of a collection of named objects with a relationship specified by axioms. The Z notation includes a mechanism for defining schemas[1] and combining them in various ways defined by a schema calculus. The partition

---

[1] *Some authors have used the plural schemata instead of schemas.*

provided by schemas allows large specifications to be built up in stages. Schemas can have generic parameters and there are operations in the Z notation for creating instances of generic schemas.

A number of introductory books about the Z notation have been published recently. Diller [Dillr90] introduces formal methods for specifying software systems in the Z notation. The book contains extensive examples of producing formal proofs, giving several inference rules for the predicates in Z schemas. The book also contains some case studies of the Z notation. The book by Potter, Sinclair and Till [Pottr91] introduces formal specification with the Z notation in the context of computer systems and the last part of the book covers the topic for computer program development. Craig [Craig91] presents the Z notation for specifying two artificial intelligent architectures. The specifications given by Craig are fairly large and detailed. Some of the Z schemas in these specifications are not quite correct according to the standard syntax of the Z notation since they use schema names instead of the bindings of the schemas. However, this practice is quite common and does not seem to present any problems when automated tools do not analyse the schemas.

The Z notation has been applied to representing various aspects of the functional behaviour of systems. Refinement techniques for the Z notation have been developed for software systems that cover abstraction levels from an initial specification to an implementation in a programming language [King90]. The main published application of the Z notation for non software systems has been restricted to specification at one level of abstraction; in this application, the level of abstraction is chosen so that the specification is free from implementation bias and is easy to analyse.

The Z notation is widely accepted as a readable (when combined with informal text) and an expressive notation for describing state in terms of data types. But it has been reported that the Z notation is less useful for describing concurrent actions and the timing or ordering of events. There can also be difficulties with structuring descriptions in the Z notation of large systems because of the global nature of the schema definitions [Duke90, Duke92]. A number of variations of the Z notation have been proposed to 'improve' its ability to represent the behaviour of systems.

A formal denotational semantics for the Z notation is presented by Spivey [Spiv88], although the semantics is considered by some not to be sufficiently formal [Saal92] and work is continuing to give a completely formal version for the Z semantics [Brien91, ZitBS91].

2.2 Related Applications of the Z Notation

Object orientation in the Z notation was discussed in a workshop [ZipOb91, Carr92] and the participants came to the conclusion that the Z notation supports objects as a language feature, but is not an object oriented language according to Wegner's classifications.

Hall describes some methods to specify systems using the object oriented approach with the Z notation [Hall90]. The usual approach with the Z notation is to specify a system as a state machine, with schemas describing parts of the state and the operations that change the state. The object oriented approach divides a system into objects, each with its own set of operations. Examples of different styles of using the Z notation in an object oriented approach to specifications are found in the ZIP report [ZipOb91]. An objected oriented version of the Z notation is used to specify the behaviour of concurrent systems by modelling process behaviour [Sch90]. There are some similarities with the methods developed by Hall and the approach used in Chapter 5 for describing sites in a replicated database system.

The terms *decomposition* and *refinement* sometimes occur in descriptions of design processes involving the Z notation. The Z notation can be used to specify the functional behaviour of systems in terms of abstract operations performed by the systems. A design process will decompose, or break up, these operations into less abstract operations. Also, as part of a design process, the abstract data types are refined, or reified, into more concrete data types used by different implementations [Spiv89A, Dillr90, Pottr91].

The Z notation has been integrated into several design processes. These processes include an information systems type of environment [Swat92], a SSADM design process [Polac92], and the Yourdon method of specification [Semm91].

In the paper by Duke and Smith [Duke89], the Z notation is used to capture liveness properties of a communications protocol. The paper compares the specifications of liveness in the standard Z notation and the enhanced Z notation which includes temporal logic operations.

The book *Specification Case Studies* edited by Ian Hayes [Hayes87] has a number of case studies of specifications written in the Z notation. The book contains several studies in similar application areas to the three presented in this thesis, however there is no direct connection with the studies in the book and those used in this research. One of the aims of using Z schemas in the case studies in the book [Hayes87] is to specify systems at an abstraction level free from any bias towards an implementation, this is either to allow novel implementations to be considered or to provide user documentation that is not obscured by the complications of the implementation of the system.

Morgan presents some non constructive descriptions in the Z notation of properties required of a communications system [Morg83] that are similar to the descriptions in the book by Hayes [Hayes87].

Woodcock and Loomes present an extensive case study of a telephone exchange [Wood88]. The state of the exchange is similar to that used by Morgan. A number of theorems and preconditions are derived by Woodcock and Loomes for their case study.

Zave and Jackson describe their work on techniques for specifying a switching system in the paper [Zave92] . The switching system described by Zave and Jackson is a small PBX that has a large number of features. The specifications generated for this PBX incorporate a state that is based on the kinds of connections and is an extension of that used by both Morgan, and Woodcock and Loomes.

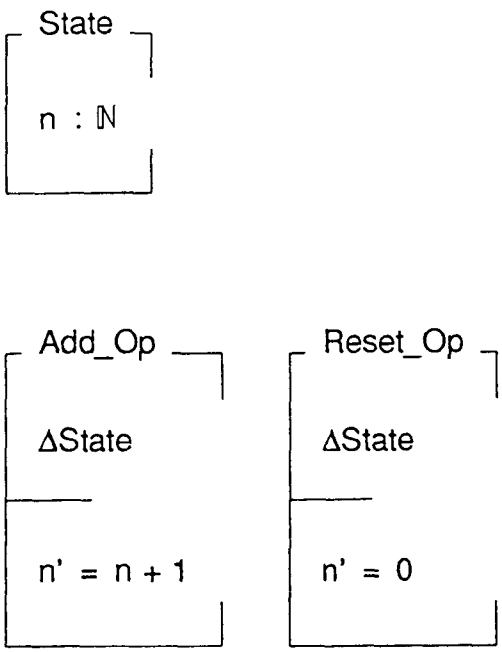
2.3 Unified Descriptions in the Z Notation

The following example of a simple counter illustrates the differences in the isolated approach commonly taken with the Z notation for specification and the integrated approach taken in this thesis for creating unified description at a design stage.

Assume that the behaviour of the counter is controlled by two operations **add** and **reset**. The **add** operation adds one to the value stored by the counter and the **reset** operation causes the counter to store the value zero.

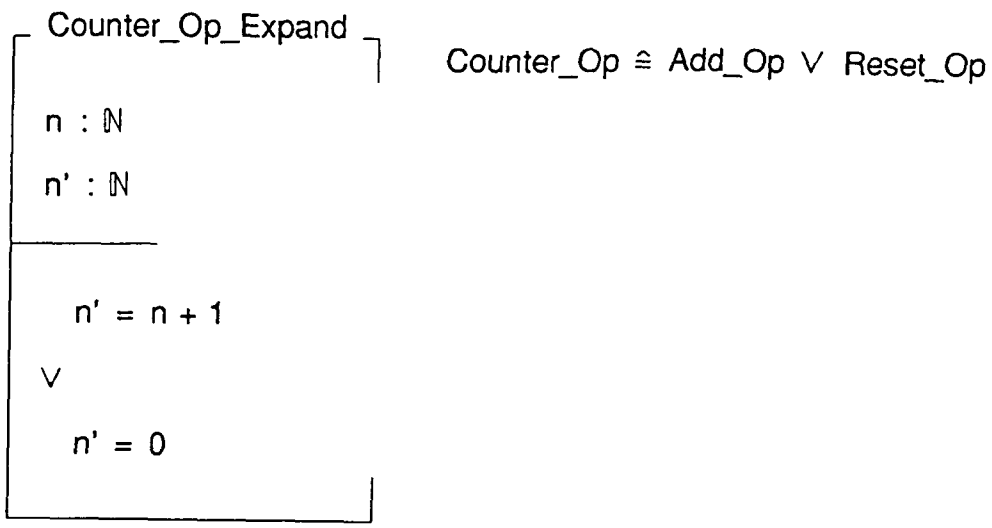
Figure 2.1 shows the Z schemas that describe the behaviour of the counter in the form of two isolated operation schemas.

Figure 2.1 Isolated Operation Schemas



Note that no information is given about what determines which operation is performed. Since the only two operations performed by the counter are **add** and **reset**, a total description of the counter is formed by the disjunction of the two operation schemas. This is shown in Figure 2.2 with the expansion of the resulting schema.

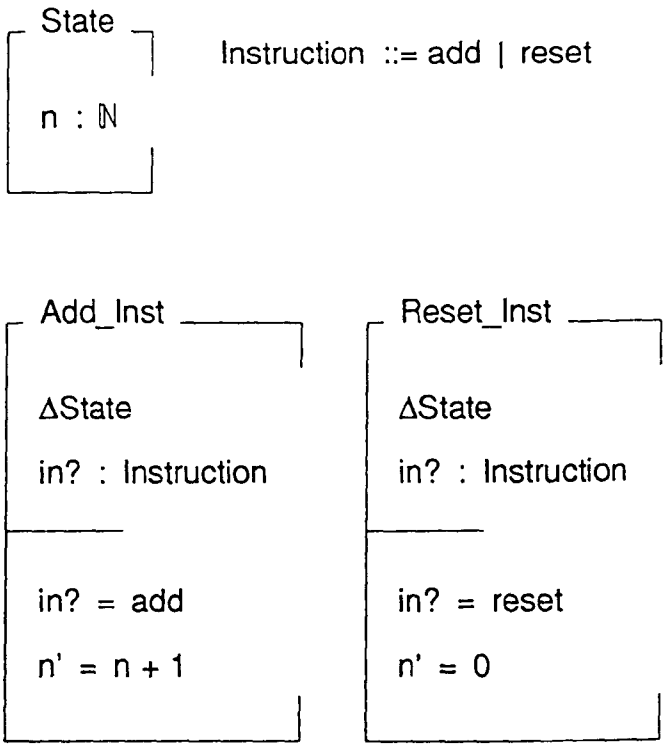
Figure 2.2 Combination of Isolated Schemas



The behaviour described by the schema Counter_Op is that the new value held by the counter is either one more than the old value or zero. This is correct but it does not indicate what determines which response. The informal method of selecting the operations is lost.

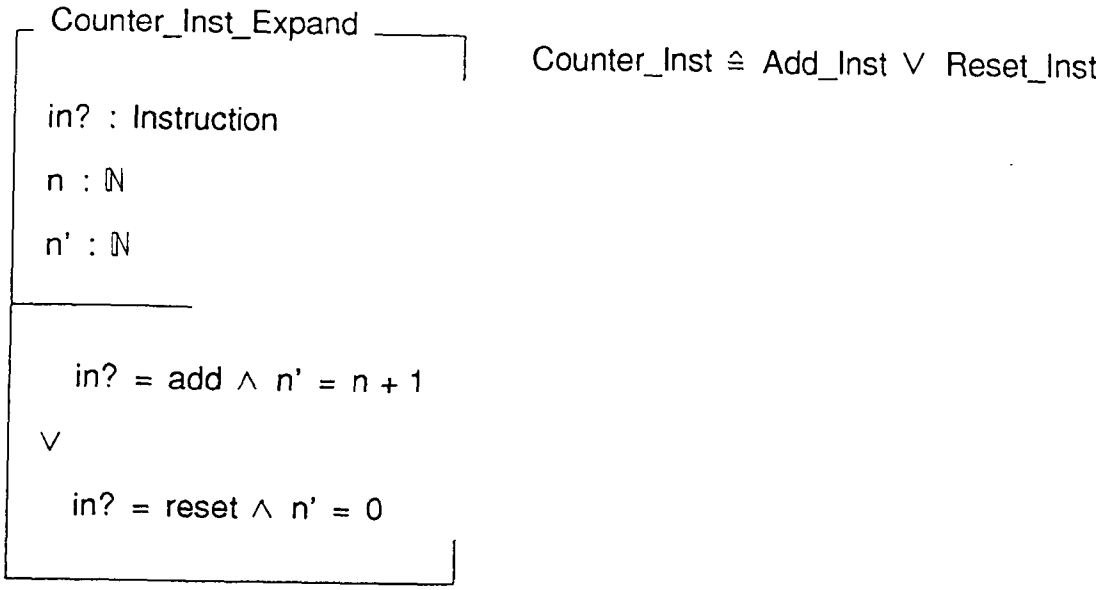
The approach taken in this thesis is to include the information that determines the operation within the operation schema. Figure 2.3 shows the equivalent operation schemas for the example of the simple counter.

Figure 2.3 Unified Operation Schemas



This time the disjunction of the operation schemas results in the schema Counter_Inst in Figure 2.4 and the expansion indicates that the behaviour of the counter is still deterministic.

Figure 2.4 Combination of Unified Schemas



2.4 CADiℤ

The schema device of the Z notation is awkward to produce satisfactorily with standard word processing packages. Several computer tools [ZipCt91] are available for type setting documents that incorporate Z schema boxes.

The Z notation includes a rigorously defined syntax and set of type rules. Checking that the rules of the Z notation are obeyed is very laborious and prone to error if performed manually. Several computer tools are available that perform syntax and type checks on documents written in the Z notation [ZipCt91].

The computer tool employed in this thesis is CADiZ (Computer Aided Design in Z), which is a suite of computer tools to check and type set specifications written in the Z notation [Jord91]. The interactive mode of CADiZ allows some properties of specifications to be investigated by displaying the expansion of schemas and deriving their signatures.

CADiZ operates in a UNIX environment and incorporates the *troff*[2] family of computer tools to produce typeset documents that are printed on paper or viewed on a screen.

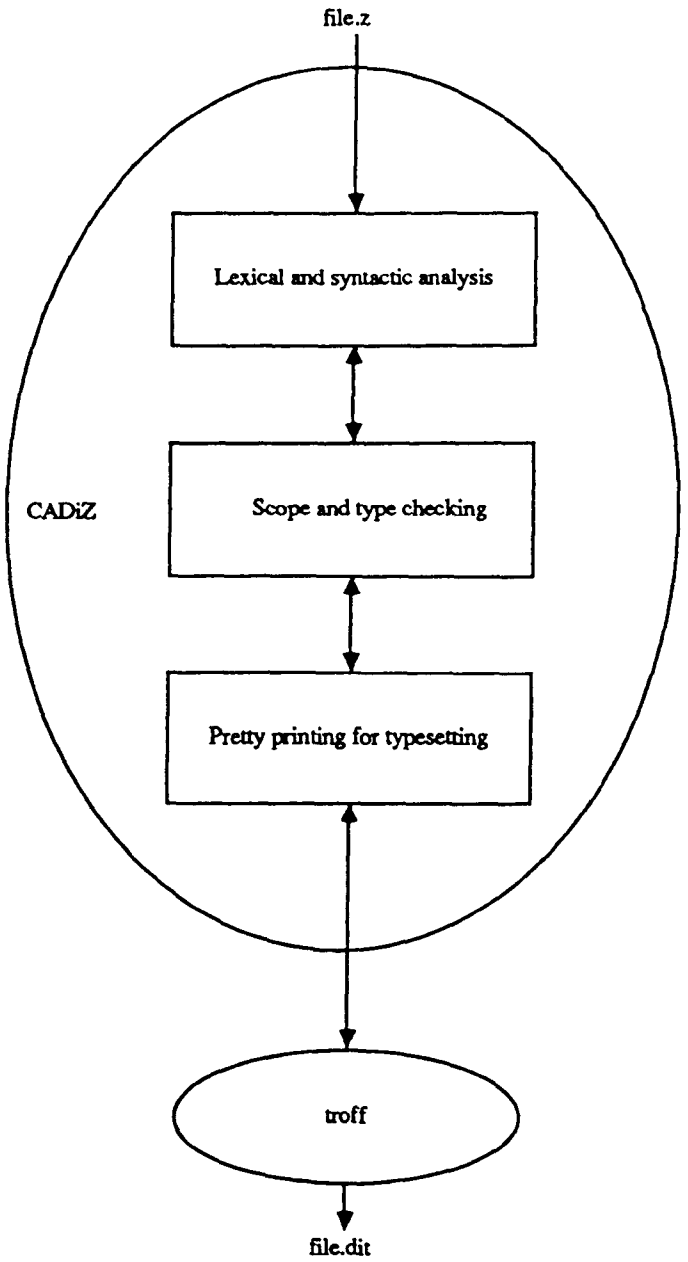
---

[2]*Troff is a text processor for the UNIX computer system that formats text for printing.*

Figure 2.5 is based on a figure in a paper by Jordon, McDermid and Toyn [Jord91]. Figure 2.5 illustrates the internal structure of CADiZ and shows the main three phases of operation.

All the schemas included in this thesis are produced using CADiZ and, unless otherwise stated, do not produced any error reports. The syntax of the Z notation used by CADiZ is reproduced in Appendix E from the *CADiZ Reference Manual* [CADiZ91].

Figure 2.5 Internal Structure of CADiZ



2.5 Summary

This thesis builds on the extensive work carried out in the development of the Z notation and formal techniques in the design of complex systems. Later chapters will apply the Z notation to the new area of replicated database systems and use the Z notation in a way more appropriate to the process of design, while keeping to the syntax and semantics of the standard version of the Z notation.

Chapter 3
An Implementation of a Communications Network

A communications network represents a system that can be easily understood, while being sufficiently complex to reveal the difficulties in unambiguously stating both the properties and implementation.

In this chapter the implementation of a simple communications network is expressed in the Z notation. The operation schemas are defined in a constructive style. The implementation is shown to possess the basic properties of a communications network, plus additional properties that result from the method of implementation.

The design in Section 3.3.7 is a schema equation formed by the disjunction of operation schema terms. Each term represents the behaviour of a network operation. All the schema terms use the same state variables and the disjunction is completely deterministic because of the disjointness of the preconditions of the schema terms.

The study presented in this chapter raises a number of questions about the application of the Z notation in a design process of complex systems. Section 3.7 discusses the main styles adopted for using the Z notation in this chapter and some of the important questions about the usefulness of the Z notation for this application.

One of the questions discussed is how best to represent concurrent activities with the Z notation without relying on informal interpretations or reducing the intelligibility of the represen-

tations. The technique found to be most successful for representing concurrent activities is to describe a system as a disjunction of operation schemas that are written in a non constructive style that does not exclude multiple operations from occurring simultaneously. This is discused in Appendix B.

This chapter represents one stage of a design process. Several stages are required before a final implementation can be reached. Appendix B contains an example of the form of an implementation for the next stage and contains examples of the three data refinement rules that must be checked.

This study illustrates the progression of the properties from informal statements to mathematical predicates and theorems.

Section 3.1.2 contains an informal description of the properties required of a simple communications network.

The safety properties stated informally in Section 3.1.2 are given formal interpretations in Section 3.4. The properties are stated to be the consequences of theorems that have implementations as the antecedents. The implementation also give rise to several additional properties, these additional properties are described in Section 3.5.

Most of the properties discussed in this chapter are safety properties. One of the ways in which liveness properties can be expressed in the Z notation is discused in Section 3.6.

Table 3.1 summarises the elements of the design in this chapter and refers to their associated sections.

Table 3.1 Summary of Design Stage in Chapter 3

Section	Description
3.1.2	Informal specification of the safety and liveness properties used in the study of a communications network.
3.2 - 3.3	Description in the Z notation of an implementation of a communications network.
3.4	Formal expression of the safety properties and their verification. Theorems 3.1 to 3.5.
3.5	Additional properties implied by the implementation. Theorems 3.6 to 3.9.
3.6	Formal expression of the liveness properties and their verification. Theorems 3.10 to 3.15.

3.1 Specification and Implementation

3.1.1 Initial Description

For the purposes of this chapter, a communications network is a system that enables subscribers to send and receive data.

In the network there are a number of subscribers connected to the network who wish to exchange data. Any subscriber can transmit data to any other subscriber and the data are received intact by the intended subscriber only. Two subscribers exchange data in the form of a conversation and conversations between different pairs of subscribers can occur simultaneously. Privacy of data is vital and a third subscriber should not be able to eavesdrop on a conversation between two other subscribers.

The destination subscriber must be able receive data before a conversation can take place. One of the reasons for a subscriber not being able to receive data is that the destination subscriber is disconnected from the network, hence inaccessible to any originating subscriber.

Integrity of data must be maintained so data must not be duplicated or lost by the network.

3.1.2 Informal Statement of Properties used in this Study

The safety properties addressed in this study are:

1 Each conversation has two subscribers.

2 It is impossible for a third subscriber to receive data destined for the second subscriber of a conversation.

3 Subscribers can be busy.

4 Subscribers can be inaccessible.

5 Data are received at most once.

The liveness properties of the behaviour of a network addressed in this study are:

1 Operations are eventually executed by the system if their preconditions are satisfied.

2 All data sent will eventually be received.

The list of properties impose requirements on the network without indicating how to achieve them. This is characteristic of a property oriented specification.

3.1.3 Informal Description of the Z Implementation

This section contains an informal description of a communications network in terms of operations performed on the network.

The subscribers, or users, of the network are people who can use it. Subscribers do not have to be connected to the network, but are capable of being connected. For subscribers to be connected they have to **join**[1] the network, subscribers who have not joined the network are inaccessible to other subscribers. Similarly, subscribers who have joined the network can **leave** and once again be inaccessible to other subscribers.

The protocol for exchanging data between two subscribers is that the originating subscriber must first **call** the destination subscriber before data can be transmitted. For a call to be suc-

---

[1] *In the following description a bold type face is used for the names of the operations that are defined later in the Z notation.*

cessful there must be a free pair of communications paths to carry the data through the network between the two subscribers.

Once a call has been set up, either subscriber can **send** data for the other subscriber to **receive**.

When all the data have been transmitted and received by both subscribers, either subscriber can **clear** the communications paths.

3.2 Data Types and Invariant

3.2.1 Given Sets

In CADiℤ, the file *toolkit* contains all the operations listed in the tool kit described by Spivey in the book *The Z Notation: A Reference Manual* [Spiv89A]. The statement

Import toolkit

makes these operations available. Two given sets are used in the schemas in this chapter.

[SUB, WORD]

The only other data types assumed are those contained in the mathematical tool kits provided automatically by CADiℤ. The source and destination of data transmitted over the network are represented by the given data type SUB and the data transmitted over the network are represented by the given type WORD in subsequent schemas.

Proof Obligation for the Consistency of Given Sets

A potential problem with given sets is that it is possible to introduce inconsistencies into the mathematical models. One method of guaranteeing that given sets have at least one feasible solution is to show that the given sets can be represented by 'known' data types that do not introduce any inconsistencies. This does not imply that the given sets have characteristics of the known data types in addition to those specified in the model. The known data types simply demonstrate that the given sets have at least one solution.

Because both SUB and WORD are used only to discriminate between elements in the sets, a consistent model is constructed if the set of natural numbers is substituted for SUB and if the set WORD is replaced by the set of all possible sequences of ASCII characters. These are not the only possible representations and only serve to indicate the consistency of the description of the communications network.

The data transmitted between subscribers is represented by the a sequence of elements from the given set WORD in the syntactic definition below:

Package == seq WORD

3.2.2 Free Data Types

The operations allowed in the communications network are represented by the free data type given below:

Operation ::= join | leave | call | clear | send | receive

Proof Obligation for the Free Type Definition of Operation

The data type definition for Operation contains six branches. All the branches are non recursive, i.e. do not include a reference to Operation. Since the definition is non recursive, the free type Operation is consistent [Arth92].

The calls between subscribers in the communications network use the concept of paths between the subscribers. These paths have the status of either *established* or *free*, represented by the identifiers in the free type data definition below.

Path_Status ::= Established | Free

The free type called Path_Status represents the two options of Established and Free.

Proof Obligation for the Free Type Definition of Path_Status

The free type definition Path_Status contains two non recursive branches and is therefore consistent.

An operation performed on the communications network is either successful or unsuccessful. These possibilities are represented by a free type definition below:

$$Error\_Status ::= Okay \mid Error$$

Proof Obligation for the Free Type Definition of Error_Status

The free type definition Error_Status has two non recursive branches labelled with the Okay and Error identifiers, and is therefore consistent.

Three types of messages are modelled in the implementation of a communications network. One type represents the actions of subscribers joining and leaving the network. A second type represents the actions of setting up and clearing down calls through the network. The final type represents the sending and receiving data through the network.

$$\begin{aligned} Message ::= \\ & member \ll (SUB \times \{join, leave\}) \gg \mid \\ & setup \ll ((SUB \times SUB) \times \{call, clear\}) \gg \mid \\ & active \ll (((SUB \times SUB) \times Package) \times \{send, receive\}) \gg \end{aligned}$$

The data type Message represents all the six operations in three branches. The operations join and leave are included in the branch identified by member. Tuples of the branch member are pairs consisting of an element with type SUB and either a join operation or a leave operation identifier.

The two operations call and clear are contained in the branch identified by setup. Members of the branch setup are tuples containing pairs of the type SUB and either the call operation or the clear operation identifier.

Each package of data is sent by a subscriber using a send operation and is received by another subscriber by a receive operation. These operations are included in the branch labelled active. Members of the branch active are tuples that contain a pair of elements of the type SUB, a data Package, and either the send operation or the receive operation identifier. Note that each message value identifies an operation as well as the parameters for that operation.

Proof Obligation for the Free Type Definition of Message

None of the branches in the definition of Message are recursive, hence the free type definition is consistent.

3.2.3 Network State Schema

The schema Network defines the common data variables included in subsequent schemas and provides a description of the state of the system.

Network

$$\begin{aligned} &Present : \mathbb{P}\ SUB \\ &Conversation : (SUB \times SUB) \rightarrow seq\ Package \\ &Path : (SUB \times SUB) \rightarrow Path\_Status \end{aligned}$$

Since the components *Conversation*[2] and *Path* are defined as total functions, any operations on them must contain the characteristics of functions.

[2] *Component identifiers of schemas are expressed in italics in the informal text using troff formatting commands. A different method of highlighting the components is to use* CADiZ *to express the component identifiers in a different font, for example* Network.Conversation *and* Network.Path. *The positions of all schema references are identified by* CADiZ *in its interactive mode to help checking the text. However, using* CADiZ *to highlight schema components has the disadvan-*

The schema Network is the invariant of the implementation of a communications network. The state space is all combinations of mappings that conform to the invariant of the properties of functions. The invariance of the characteristics of functions is a proof obligation. In this case, the discharge of this obligation is obvious for all the operation schemas, however, a brief sketch is given to emphasise the importance of discharging such obligations.

Initial State of the Network

It is important to demonstrate that there is at least one feasible state for the system. This is done by constructing a schema that specifies the conditions of an initial state. The schema Initial_State defines a valid state of the system.

Initial_State

Network'

$\forall$ u1, u2 : SUB •

Conversation' (u1, u2) = $\langle\rangle$ $\wedge$ Path' (u1, u2) = Free

Present' = $\varnothing$

The schema Initial_State follows the convention of using decorated variable names for defining the initial state [Dillr90, Pottr91].

Proof Obligation for the Initial State

The invariants given by the schema Network' are obviously true. That is, *Conversation'* and *Path'* are functions, and *Present'* is a set.

---

tage of having to qualify the component identifiers with the schema names which makes some sentences appear awkward.

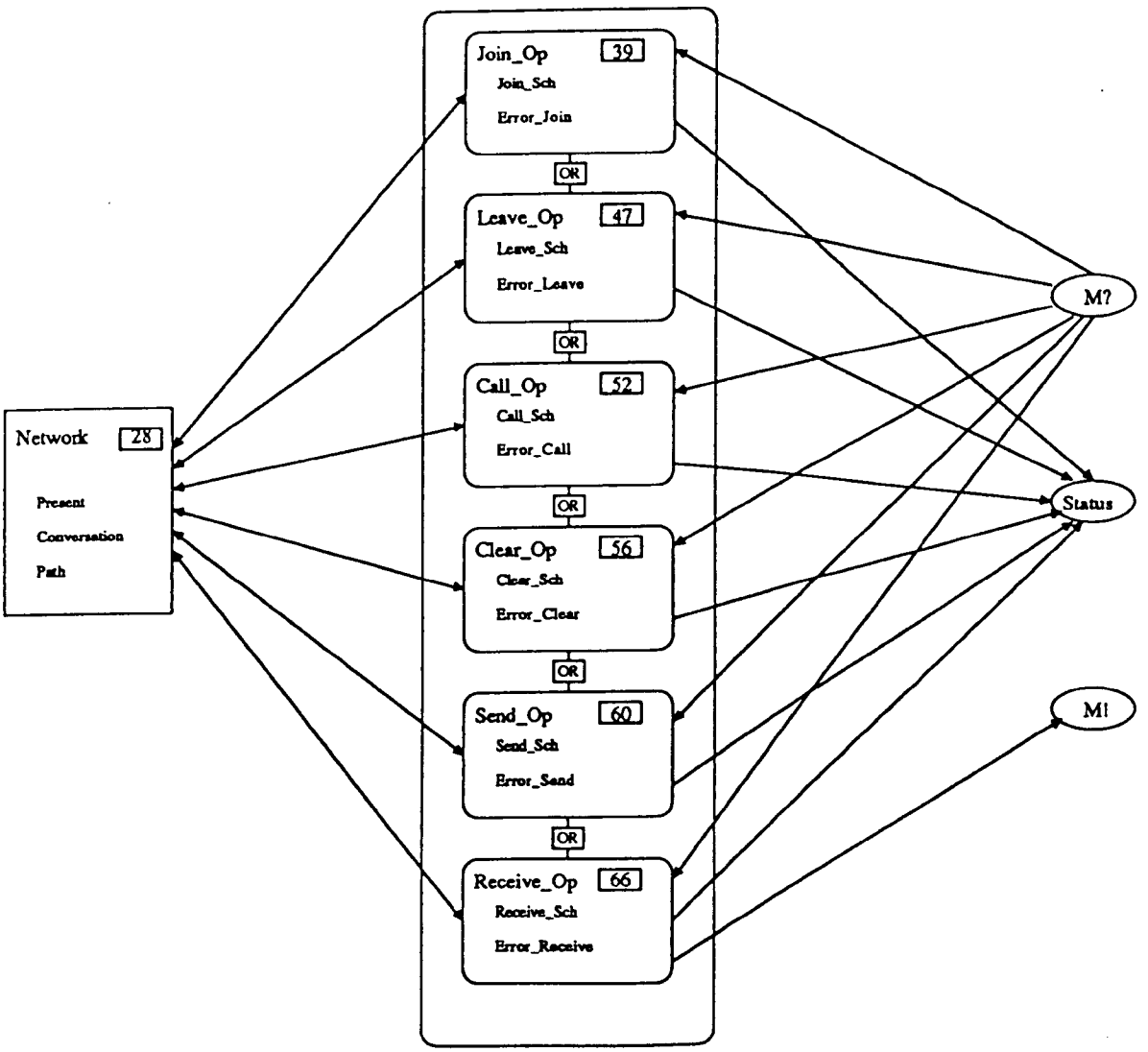
Diagrams of Interactions between Schemas

Figure 3.1 provides an overview of the interactions in the complete implementation in the Z notation. Each operation schema is given a more detailed diagrammatic representation following the associated schema definitions. The diagrams are to help understanding the formal descriptions in the Z notation by showing the relations between the schemas. The diagrams are not intended either to add to the formality of the descriptions or to provide interpretations of their behaviour. Using the diagrams to add information to the schemas can introduce ambiguity, hence such diagrams should be used with care.

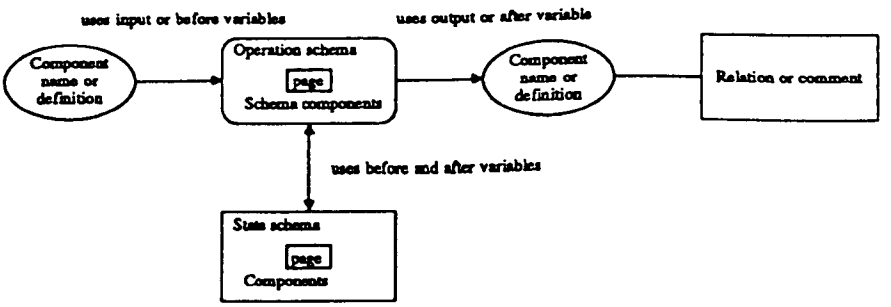
Diagrams are useful for providing references to the schemas in a large system that are inevitably dispersed over several pages. The diagrams also provide a picture of the interactions between the parts of a complex description, such interactions are often difficult to acquire directly from the schemas. Moreover, the process of drawing diagrams that are based on Z schemas is very useful for identifying slips such as missing components from the schemas.

Figure 3.1 is typical of the interaction diagrams contained in this thesis. Schemas that describe the state of the system are shown as rectangular boxes, whereas operation schemas as shown as boxes with rounded corners. Elliptical boxes are used to identify components of schemas. Relations between two schemas and between a schema and a component are described by comments contained in rectangular boxes. Arrows are used to indicate whether the before, after or both versions of components are used. An input variable is considered to be a before variable and an output variable is considered to be an after variable.

Figure 3.1 Overview of the Network Implementation



Key



3.3 Network Operations

3.3.1 The Join Operation

Before subscribers can send or receive data over the network they must first join the network.

Join_Sch

$$
\begin{array}{l}
\Delta Network \\
M? : Message \\
Status' : Error\_Status \\
\hline
\exists\, sub : SUB;\ op : Operation \mid \\
\quad M? = member(sub, op) \land op = join \land sub \notin Present \bullet \\
\qquad Conversation' = \\
\qquad Conversation \oplus \{u : SUB \bullet (u, sub) \mapsto \langle\rangle\} \oplus \\
\qquad \{u : SUB \bullet (sub, u) \mapsto \langle\rangle\} \land \\
\qquad Path' = \\
\qquad Path \oplus \{u : SUB \bullet (u, sub) \mapsto Free\} \oplus \\
\qquad \{u : SUB \bullet (sub, u) \mapsto Free\} \land \\
\qquad Present' = Present \cup \{sub\} \land Status' = Okay
\end{array}
$$

All Network variables changed

The schema Join_Sch represents the operation of subscribers joining the communications network. This schema describes one aspect of the dynamic behaviour of the communications network. The schema is split into two parts; the upper part declares the data types, as in the schema Network, and the lower part defines the predicates associated with the schema. The predicate part of schema defines the preconditions that must be satisfied before any change of state and the postconditions that apply to the after state. There is no formal separation of the preconditions and postconditions in the predicate part of schemas.

The predicate part of a schema must be true for each change of state.

The postconditions define the after state of the system and are defined in terms of values of the after variables. Any after variable not defined in the schema can take any value consistent with its declared data type.

The declaration part of the schema uses the notation ΔNetwork which has the effect of declaring all the data types in the schema Network both in their decorated and plain forms. Placing a schema identifier in the declaration part of a schema is called *schema inclusion.* The plain forms represent the variables before the operations are performed and the decorated variables (i.e. the same identifiers but postfixed with apostrophes, also called single dashes or primes) represent the variables after the operations have been performed; subsequently called after variables. The Join_Sch also uses the convention of ending each of the the identifiers representing an input variable with a question mark, as in the case of the *M?* variable [Spiv89A, Dillr90, Pottr91].

The *M?* input represents a join operation, hence the elements of the tuple in the member branch of the data type Message are extracted by equating *M?* with the two existentially quantified variables *sub* and *op* which represent the joining subscriber and operation respectively. The two other branches of the data type definition of Message do not meet this precondition. The other preconditions of the schema Join_Sch are that the operation is a join operation and the value of the subscriber is not a member of the set *Present,* which is the set of all subscribers that have joined and not left.

The function *Conversation* is updated so that all the tuples in its domain that have an element of the same value as the *sub* variable map to the empty sequence. Similarly, the function *Path* is updated so that the tuples in its domain that refer to subscribers with the same value as *sub* map to the value Free.

The set *Present* is updated so that on completion of the operation it contains the subscriber value represented by *sub*.

Finally, the *Status'* variable is bound to the value Okay to indicate a successful completion of an operation.

Preconditions for Join_Sch

In this study the preconditions are used for two main reasons. The first reason is that the preconditions allow the implementation to be checked that it is defined for all conditions. The second reason is that the operations are verified to be disjoint; it is not possible for two operations to respond when in the same state and receive the same input.

Preconditions are formed by moving all the declarations of after variables (identified by dashes as the final character of their names) and output variables to the predicate part of the schema by existential quantification [Wood89A, Dillr90]. The preconditions define the conditions necessary for a change of state and the possible changes of state are identified by an operation schema. The after variables and output variables reflect this change of state, however, their actual values do not affect when the operation can be performed as the after variables and data variables are determined by the schema itself, hence they are existentially quantified. The preconditions are only concerned with the after variables and output variables in as far as the schema invariants are not violated, that is, they define a valid state. A valid state is one in which the invariant evaluates to true.

The preconditions of the schema Join_Sch are represented by the schema below which is created using the *pre* operator [Spiv89A, Dillr90].

Pre_Join_Sch $\widehat{=}$ pre Join_Sch

The preconditions of the Join_Sch are made explicit in the schema Pre_Join_Sch_Expand below.

Pre_Join_Sch_Expand

$$
\begin{array}{l}
Present : \mathbb{P}\ SUB \\
Conversation : (SUB \times SUB) \rightarrow seq\ Package \\
Path : (SUB \times SUB) \rightarrow Path\_Status \\
M? : Message \\
\hline
\exists\ Present' : \mathbb{P}\ SUB; \\
\quad Conversation' : (SUB \times SUB) \rightarrow seq\ Package; \\
\quad Path' : (SUB \times SUB) \rightarrow Path\_Status; \\
\quad Status' : Error\_Status \bullet \\
\qquad \exists\ sub : SUB; op : Operation \mid \\
\qquad\quad M? = member\ (sub, op) \land op = join \land sub \notin Present \bullet \\
\qquad\qquad Conversation' = \\
\qquad\qquad Conversation \oplus \{u : SUB \bullet (u, sub) \mapsto \langle\rangle\} \oplus \\
\qquad\qquad \{u : SUB \bullet (sub, u) \mapsto \langle\rangle\} \land \\
\qquad\qquad Path' = \\
\qquad\qquad Path \oplus \{u : SUB \bullet (u, sub) \mapsto Free\} \oplus \\
\qquad\qquad \{u : SUB \bullet (sub, u) \mapsto Free\} \land \\
\qquad\qquad Present' = Present \cup \{sub\} \land Status' = Okay
\end{array}
$$

Simplifying the Preconditions

The CADiZ tool in its interactive mode expands schema expressions, such as Pre_Join_Sch, to display the schemas in full, which can help simplifying the preconditions. See Appendix B.1 for examples of the outputs produced by CADiZ.

The precondition schema Pre_Join_Sch_Expand is simplified to give the schema Pre_Join_Sch_Simple by repeated application of the one point rule [Wood89A].

Pre_Join_Sch_Simple

Network

M? : Message

$\exists$ sub : SUB • M? = member (sub, join) $\wedge$ sub $\notin$ Present

The fact that the simplified version of the preconditions of the schema Join_Sch is true in precisely the same conditions as the rudimentary form of the preconditions given by the schema Pre_Join_Sch is expressed in the schema Simplified_3_1 below.

Simplified_3_1 $\triangleq$ Pre_Join_Sch $\Longleftrightarrow$ Pre_Join_Sch_Simple

Emphasising earlier comments about preconditions; the equivalence comes about because preconditions are only concerned with the existence of valid next states, not the actual next state.

Restrictions of CADiℤ

The equivalence between the rudimentary preconditions and the simplified preconditions is stated in the form of a schema definition to allow CADiℤ in its interactive mode to expand the schema definition. This is particularly useful if one of the schema terms includes the *pre* operator.

The expansion of the schema produced by CADiℤ is contained in Appendix B.1.2. Using schema declarations for equivalence relations in this manner eases the comparisons between the schemas produced by CADiℤ.

Although CADiℤ can be used to expand schemas and display some expansions of preconditions, it cannot be used to simplify the preconditions, which means that all the simplifications of the preconditions in this chapter were performed manually using simple cut and paste commands on a word processor. See Appendix B.2 for an example of the simplification process.

Strengthening Schema Definitions of the Join Operation

The initial definition of the join operation in the schema Join_Sch is a partial definition because it leaves the behaviour unspecified when the join operation is attempted and the subscriber is already connected to the network. The schema Error_Join rectifies this by making the definition of the join operation total for all possible conditions for that operation.

Error_Join

$$\Xi Network$$
$$M? : Message$$
$$Status' : Error\_Status$$

$$\exists sub : SUB; op : Operation \mid$$
$$M? = member(sub, op) \land op = join \land sub \in Present \bullet$$
$$Status' = Error$$

The schema Error_Join increases the robustness of the join operation by including a precondition that is true if the subscriber is a member of the set *Present*.

The notation ΞNetwork represents the declaration of the variables in the schema Network in both the decorated and plain forms [Spiv89A, Dilli90, Pottr91], with the additional predicate that all the decorated terms are equal to the plain terms, i.e. there is no change of state identified by those variables declared in the schema Network.

The only change of state is represented by the *Status'* variable in the schema Error_Join.

Preconditions for Error_Join

The preconditions are defined by the schema equation:

Pre_Error_Join ≙ pre Error_Join

The above schema definition can be expanded using CADiℤ. Unfortunately, CADiℤ does not existentially quantify the after variables for schemas declared using the Ξ schema name declaration. See Appendix B.1.3 for an example. Using the explicit form for no change of state in the predicate part of the schema given in *The Z Notation: A Reference Manual* [Spiv89A] is also not expanded by CADiℤ.

The schema Pre_Error_Join_Expand is written out in full below.

Pre_Error_Join_Expand

$$
\begin{array}{l}
\text{Present} : \mathbb{P}\ \text{SUB} \\
\text{Conversation} : (\text{SUB} \times \text{SUB}) \rightarrow \text{seq Package} \\
\text{Path} : (\text{SUB} \times \text{SUB}) \rightarrow \text{Path\_Status} \\
\text{M?} : \text{Message} \\
\hline
\exists\ \text{Present'} : \mathbb{P}\ \text{SUB}; \\
\quad \text{Conversation'} : (\text{SUB} \times \text{SUB}) \rightarrow \text{seq Package}; \\
\quad \text{Path'} : (\text{SUB} \times \text{SUB}) \rightarrow \text{Path\_Status}; \\
\quad \text{Status'} : \text{Error\_Status} \bullet \\
\qquad \exists\ \text{sub} : \text{SUB};\ \text{op} : \text{Operation} \mid \\
\qquad\quad \text{M?} = \text{member}\ (\text{sub}, \text{op}) \wedge \text{op} = \text{join} \wedge \text{sub} \in \text{Present} \bullet \\
\qquad\qquad \text{Status'} = \text{Error} \wedge \\
\qquad\qquad (\text{Present'} = \text{Present} \wedge \text{Conversation'} = \text{Conversation} \wedge \\
\qquad\qquad\ \text{Path'} = \text{Path})
\end{array}
$$

Simplification of Preconditions

The preconditions are simplified to the schema Pre_Error_Join_Simple below.

Pre_Error_Join_Simple

Network

M? : Message

$\exists$ sub : SUB • M? = member (sub, join) $\wedge$ sub $\in$ Present

The schema Pre_Error_Join_Expand is equivalent to the preconditions of the schema Pre_Error_Join_Simple, which is defined in the schema below.

Simplified_3_2 $\triangleq$ Pre_Error_Join $\Longleftrightarrow$ Pre_Error_Join_Simple

The robust version of the join operation is described by the disjunction of the two schemas Join_Sch and Error_Join in the definition of the schema Join_Op shown below.

Join_Op $\triangleq$ Join_Sch $\vee$ Error_Join

The disjunction of the two schemas has the two effects of combining their declarations and forming the disjunction of their predicates. All declarations of the same identifiers must have the same signatures, i.e. agree in basic type as expressed in terms of the predefined types of the Z notation.

The expansion of the schema Join_Op is given in the schema Join_Op_Expand below to illustrate the amount of information represented by the declaration of the schema Join_Op above.

Join_Op_Expand

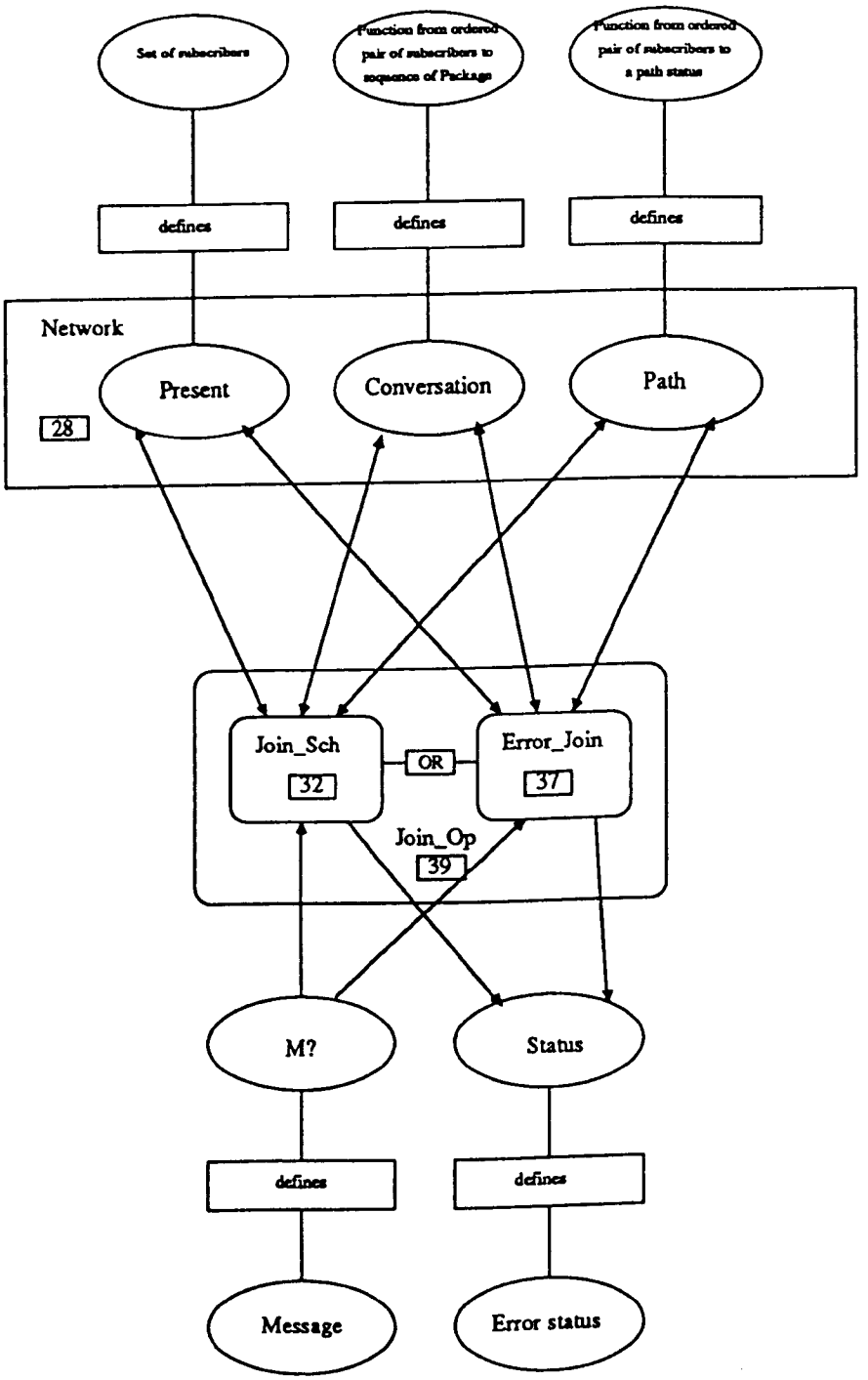
$$
\begin{aligned}
&\text{Conversation} : (\text{SUB} \times \text{SUB}) \rightarrow \text{seq Package} \\
&\text{Conversation'} : (\text{SUB} \times \text{SUB}) \rightarrow \text{seq Package} \\
&\text{M?} : \text{Message} \\
&\text{Path} : (\text{SUB} \times \text{SUB}) \rightarrow \text{Path\_Status} \\
&\text{Path'} : (\text{SUB} \times \text{SUB}) \rightarrow \text{Path\_Status} \\
&\text{Present} : \mathbb{P}\ \text{SUB} \\
&\text{Present'} : \mathbb{P}\ \text{SUB} \\
&\text{Status'} : \text{Error\_Status} \\
\hline
&(\exists\, \text{sub} : \text{SUB};\ \text{op} : \text{Operation} \mid \\
&\quad \text{M?} = \text{member}(\text{sub}, \text{op}) \land \text{op} = \text{join} \land \text{sub} \notin \text{Present} \bullet \\
&\qquad \text{Conversation'} = \\
&\qquad \text{Conversation} \oplus \{u : \text{SUB} \bullet (u, \text{sub}) \mapsto \langle\rangle\} \oplus \\
&\qquad \{u : \text{SUB} \bullet (\text{sub}, u) \mapsto \langle\rangle\} \land \\
&\qquad \text{Path'} = \\
&\qquad \text{Path} \oplus \{u : \text{SUB} \bullet (u, \text{sub}) \mapsto \text{Free}\} \oplus \\
&\qquad \{u : \text{SUB} \bullet (\text{sub}, u) \mapsto \text{Free}\} \land \\
&\qquad \text{Present'} = \text{Present} \cup \{\text{sub}\} \land \text{Status'} = \text{Okay}) \\
\lor \\
&(\exists\, \text{sub} : \text{SUB};\ \text{op} : \text{Operation} \mid \\
&\quad \text{M?} = \text{member}(\text{sub}, \text{op}) \land \text{op} = \text{join} \land \text{sub} \in \text{Present} \bullet \\
&\qquad \text{Status'} = \text{Error}) \land \text{Present'} = \text{Present} \land \\
&\text{Conversation'} = \text{Conversation} \land \text{Path'} = \text{Path}
\end{aligned}
$$

Diagram of Interactions of an Operation Schema

The interactions between the schema Join_Op and the other components in the system are shown in Figure 3.2. One of the disadvantages of using diagrams that indicate the relations between schemas by lines is that the diagrams soon become confusing for systems that share the same state. In this study all the network operations use the schema Network, but showing all six operations on the same diagram at the level of detail of Figure 3.2 would be too confusing to be of practical use. The approach adopted in this thesis is to have an overview diagram, such as Figure 3.1, and more a detailed diagram for each set of schemas for an operation, such as Figure 3.2.

Note that in Figure 3.2 double ended arrows are drawn between all three components in the schema Network and the schema Error_Join, indicating that the before and after versions of the components are used. Although the schema Error_Join does not change the bindings to the components in the schema Network, it does specify that their bindings remain unchanged.

Figure 3.2 Interactions between the Join Operation Schema and the State Schema



Note that the schema is still partial over the complete set of conditions as it does not define the behaviour in response to other operations. The interactions shown in Figure 3.2 are considered in the following proof obligation.

Proof Obligation for the Invariant of Schema Join_Op

It is necessary to ensure that the invariants for the state schema are not violated by the changes of state described by the operation schema Join_Op. Although there are no predicates in the schema Network that restrict the number of valid states, both the components *Conversation* and *Path* are defined to be total functions. This means that the functional property must not be violated. The property for a total function can be expressed for total function, F, as:

$$\{F : X \leftrightarrow Y \mid \forall x : X; y, z : Y \bullet x\ F\ y \land x\ F\ z \Rightarrow y = z\} \land (dom\ F = X)$$

CADiZ checks that functions retain their relation characteristic of the correct mapping from type X to type Y, but the deterministic characteristic for each element in the domain of a function is not checked.

The schema Join_Sch updates the functions *Conversation* and *Path* using the functional override operator to replace elements in their domains with exactly one element in their ranges, hence maintaining the functional property. This can be seen in the general case of the functional override operator maintaining the functional characteristics if it is applied correctly. The type of the functional override is given as [Spiv89A]:

$$(X \pfun Y) \times (X \pfun Y) \rightarrow (X \pfun Y)$$

However, from the law that

$$dom(f \oplus g) = (dom\ f) \cup (dom\ g)$$

the type of the functional override can be expressed for total functions as:

$$(X \rightarrow Y) \times (X \pfun Y) \rightarrow (X \rightarrow Y)$$

The changes introduced to the functions *Conversation* and *Path* are given in terms of sets of mappings that correspond to partial functions of the correct type, hence the total functional properties of both functions are not violated by the schema Join_Sch.

The schema Error_Join does not change either of the functions *Conversation* and *Path*, hence cannot inviolate their characteristics.

Finally, the schemas Join_Sch and Error_Join have mutually exclusive preconditions, hence there can be no interference between the two schemas when they are combined to form the schema Join_Op.

Preconditions for Join_Op

The precondition for the schema Join_Op is described by the schema Pre_Join_Op below.

$$\text{Pre\_Join\_Op} \mathrel{\widehat{=}} \text{Pre\_Join\_Sch} \lor \text{Pre\_Error\_Join}$$

The schema Pre_Join_Op is simplified to the schema Pre_Join_Op_Simple defined below.

Pre_Join_Op_Simple

$$\text{Network}$$
$$M? : \text{Message}$$

$$\exists\, \text{sub} : \text{SUB} \bullet M? = \text{member}(\text{sub}, \text{join})$$

The equivalence of the precondition schemas is represented by the following schema Simplified_3_3 :

$$Simplified\_3\_3 \mathrel{\widehat{=}} pre\ Join\_Op \Leftrightarrow Pre\_Join\_Op\_Simple$$

The simplification of the preconditions for the join operation is given in Appendix B.2.

Other Expressions of Simplifications

The above is not the only form of equivalence relation between the simplified and rudimentary preconditions. Another expression of the preconditions is defined as the disjunction of the preconditions of the individual schemas in the definition of the schema Join_Op and is declared as the schema Simplified_3_4 below.

$$Simplified\_3\_4 \mathrel{\widehat{=}} pre\ Join\_Sch \lor pre\ Error\_Join \Leftrightarrow Pre\_Join\_Op\_Simple$$

A further possibility is using the following schema equation below.

$$Simplified\_3\_5 \mathrel{\widehat{=}} Pre\_Join\_Op \Leftrightarrow Pre\_Join\_Op\_Simple$$

A general theorem about preconditions is proved by Woodcock [Wood89A] for the disjunction of schema terms is that:

$$pre\ (Schema\_1 \lor Schema\_2) = (pre\ Schema\_1) \lor (pre\ Schema\_2)$$

The expression chosen to represent the equivalence between two forms of the preconditions will depend on which is easiest to verify.

3.3.2 The Leave Operation

The counterpart to the join operation is the leave operation.

Leave_Sch

ΔNetwork

M? : Message

Status' : Error_Status

$\exists$ sub : SUB; op : Operation |

M? = member (sub, op) $\wedge$ op = leave $\wedge$ sub $\in$ Present $\wedge$

($\forall$ u : SUB •

Conversation (u, sub) = $\langle\rangle$ $\wedge$ Conversation (sub, u) = $\langle\rangle$ $\wedge$

Path (u, sub) = Free $\wedge$ Path (sub, u) = Free) •

Present' = Present \ {sub} $\wedge$ Status' = Okay

No change in the other Network declarations

Conversation' = Conversation

Path' = Path

The two preconditions for subscribers leaving the network are that the subscriber is not involved in a conversation with any subscriber and is a member of the communications network before the operation is performed. The above schema Leave_Sch defines the necessary preconditions and postconditions for the leave operation. The postconditions are defined to be that the set *Present'* has the member with the value given by *sub* removed and the variable *Status'* has the value Okay after the operation has been completed.

Improving the Robustness

The schema Error_Leave defines the preconditions necessary to trap the conditions that would cause the leave operation to fail and the postcondition is the variable *Status'* has the value Error after the operation had been attempted.

Error_Leave

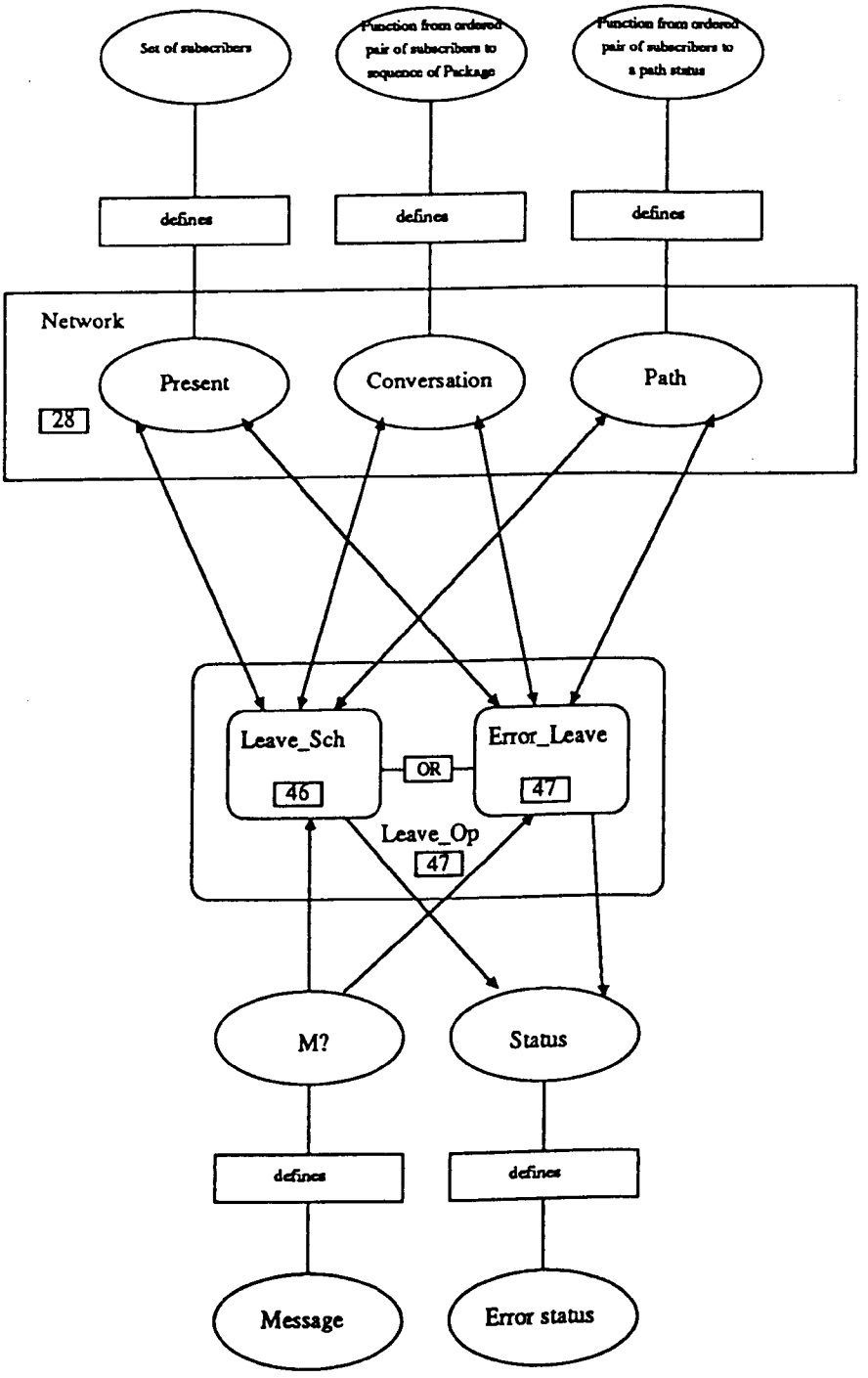
$$
\begin{array}{l}
\Xi \text{Network} \\
M? : \text{Message} \\
\text{Status}' : \text{Error\_Status} \\
\hline
\exists\, \text{sub} : \text{SUB};\ \text{op} : \text{Operation} \mid \\
\quad M? = \text{member}(\text{sub}, \text{op}) \land \text{op} = \text{leave} \land \\
\quad (\text{sub} \notin \text{Present} \lor \\
\quad \neg \\
\quad (\forall u : \text{SUB} \bullet \\
\qquad \text{Conversation}(u, \text{sub}) = \langle\rangle \land \\
\qquad \text{Conversation}(\text{sub}, u) = \langle\rangle \land \text{Path}(u, \text{sub}) = \text{Free} \land \\
\qquad \text{Path}(\text{sub}, u) = \text{Free})) \bullet \text{Status}' = \text{Error}
\end{array}
$$

A robust version of the leave operation is defined as the disjunction of the two schemas Leave_Sch and Error_Leave in the schema Leave_Op below.

$$\text{Leave\_Op} \triangleq \text{Leave\_Sch} \lor \text{Error\_Leave}$$

Figure 3.3 shows the interactions between the schema Leave_Op and the other components of the system.

Figure 3.3 Interactions between the Leave Operation Schema and the State Schema



Proof Obligation for the Invariant of the Schema Leave_Op

CADiZ has ensured that the basic types are not violated, and the functions *Conversation* and *Path* are not changed, hence the functional properties are maintained.

Preconditions for Leave_Op

The preconditions of the robust version of the operation are simplified to the schema Pre_Leave_Op_Simple below.

Pre_Leave_Op_Simple

Network

M? : Message

$\exists$ sub : SUB • M? = member (sub, leave)

The equivalence between the two expressions of the preconditions is expressed as the following schema.

Simplified_3_6 $\triangleq$ pre Leave_Op $\Leftrightarrow$ Pre_Leave_Op_Simple

3.3.3 The Call Operation

The first two operations define the effects of subscribers joining and leaving the communications network analysed in this chapter. A call path must be established between the two subscribers before any data are transmitted between two subscribers in the network. This is certainly not the only possible implementation, for example E-mail does not require any call path to be established before data transmission. Decisions at this stage in the design process have large impacts on the final implementation of the communications network and

it may be necessary to compare different implementations at this level of abstraction before deciding which one to refine further.

The schema Call_Sch below specifies the operation of setting up a call path.

Call_Sch

ΔNetwork

M? : Message

Status' : Error_Status

$\exists$ calling, called : SUB; op : Operation |

M? = setup ((calling, called), op) $\wedge$ op = call $\wedge$

calling $\in$ Present $\wedge$ called $\in$ Present $\wedge$ calling $\neq$ called $\wedge$

Conversation (calling, called) = $\langle\rangle$ $\wedge$

Conversation (called, calling) = $\langle\rangle$ $\wedge$

Path (calling, called) = Free $\wedge$ Path (called, calling) = Free •

Path' =

Path $\oplus$ {(calling, called) $\mapsto$ Established} $\oplus$

{(called, calling) $\mapsto$ Established} $\wedge$ Status' = Okay

No change in the variables Conversation and Present

Present' = Present

Conversation' = Conversation

The first terms of the above schema extract the elements of the tuple given by the value of the *M?* input message. This is done by equating *M?* to the existentially quantified tuple of the *calling* subscriber, *called* subscriber and operation *op*. The following preconditions are specified using these bound variables:

1 both subscribers are connected to the network, which is represented by both subscribers being members of the set *Present*

2 the two subscribers are not identical, which is represented by *calling* $\neq$ *called*

3 no data are in transit in either direction between the two subscribers, which is represented by both possible orders of the pairs of subscribers mapped to empty sequences by the function *Conversation*

4 there is not a path established in either direction between the two subscribers, which is represented by the function *Path* mapping both possible orders of the pairs of subscribers to the identifier Free.

After the call operation, the function *Path* is updated so that both orders of the pair of subscribers are mapped to the identifier Established.

There are no changes to the variables *Conversation* and *Present* introduced by the schema Call_Sch.

Improving the Robustness

The schema Error_Call includes the preconditions necessary to ensure that the Network state does not change if the call operation is attempted when the preconditions of the schema Call_Sch are not satisfied.

Error_Call

ΞNetwork

M? : Message

Status' : Error_Status

$\exists$ calling, called : SUB; op : Operation |

M? = setup ((calling, called), op) $\wedge$ op = call $\wedge$

(calling $\notin$ Present $\vee$ called $\notin$ Present $\vee$ calling = called $\vee$

Conversation (calling, called) $\neq \langle\rangle$ $\vee$

Conversation (called, called) $\neq \langle\rangle$ $\vee$

Path (calling, called) $\neq$ Free $\vee$ Path (called, calling) $\neq$ Free) •

Status' = Error

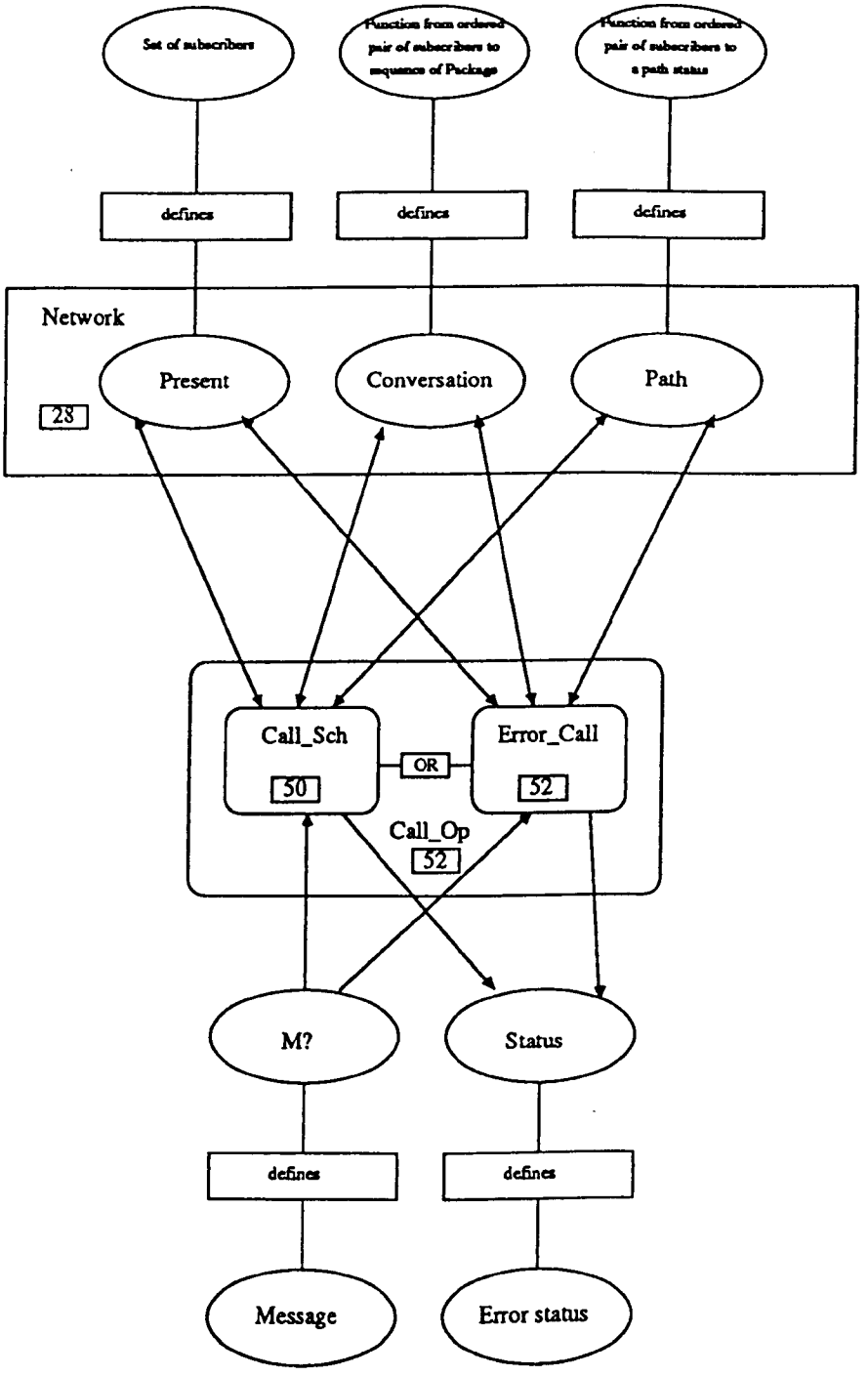
The *Status'* variable in the schema Error_Call is updated to be bound to the value Error should any of the four preconditions listed above be false.

The robust version of the call operation is represented by the schema Call_Op and is defined as the disjunction of the schemas Call_Sch and Error_Call.

$$\text{Call\_Op} \triangleq \text{Call\_Sch} \vee \text{Error\_Call}$$

Figure 3.4 shows the interactions between the schema Call_Op and the other components in the system. These interactions are considered in the following proof obligation.

Figure 3.4 Interactions between the Call Operation Schema and the State Schema



Proof Obligation for the Invariant of the Schema Call_Op

CADiℤ has ensured that the basic types are not violated.

The function *Path* is changed by the schema Call_Sch such that elements of its domain are mapped to single elements in its range, therefore maintaining the type required by the functional override operator to ensure that the resulting type of *Path* is still a total function.

The schema Error_Call does not change either the functions *Conversation* or *Path*, hence cannot violate their functional properties.

The preconditions of the schemas Call_Sch and Error_Call are mutually exclusive, hence there can be no conflict in changes of state.

Preconditions for Call_Op

The simplified preconditions of the schema Call_Op are calculated to be that defined by the schema Pre_Call_Op_Simple given below.

Pre_Call_Op_Simple

$$\begin{array}{l} Network \\ M? : Message \\ \hline \exists\, calling, called : SUB \bullet M? = setup((calling, called), call) \end{array}$$

The equivalence between the rudimentary preconditions and those derived are expressed as the schema given below.

$$Simplified\_3\_7 \triangleq pre\ Call\_Op \Leftrightarrow Pre\_Call\_Op\_Simple$$

3.3.4 The Clear Operation

The counterpart to the call operation is the clear operation.

Clear_Sch

ΔNetwork

M? : Message

Status' : Error_Status

$\exists$ calling, called : SUB; op : Operation |

M? = setup ((calling, called), op) $\wedge$ op = clear $\wedge$

Path (calling, called) = Established $\wedge$

Path (called, calling) = Established $\wedge$

Conversation (calling, called) = $\langle\rangle$ $\wedge$

Conversation (called, calling) = $\langle\rangle$ •

Path' =

Path $\oplus$ {(calling, called) $\mapsto$ Free} $\oplus$

{(called, calling) $\mapsto$ Free} $\wedge$ Status' = Okay

No change in the variables Conversation and Present

Present' = Present

Conversation' = Conversation

The schema Clear_Sch specifies the effects of the clear operation on the state of the system. In this implementation of a communications network, all the data must be received before a call is cleared.

The preconditions restrict the clearing of a call to the cases where a path has been established in both directions and that there are no data packages in transit. The effects of the clear operation are that the members in the domain for both possible orders of the pair of subscribers in the function *Path'* map to the value Free and the variable *Status'* is bound to the value Okay.

Note that there is not a precondition for the subscribers being present in the schema Clear_Sch since this condition is covered by having established paths. Also, the precondition in the schema Call_Sch that ensures that there are no data packages in transit is redundant when associated with the precondition of the call path being free. However, the preconditions are left in their original form to reduce the uncertainty about the initial condition of the network before any operations have been performed.

Improving the Rebustness

The schema Error_Clear is required for the cases when the preconditions in the schema Clear_Sch are not satisfied for the clear operation.

Error_Clear

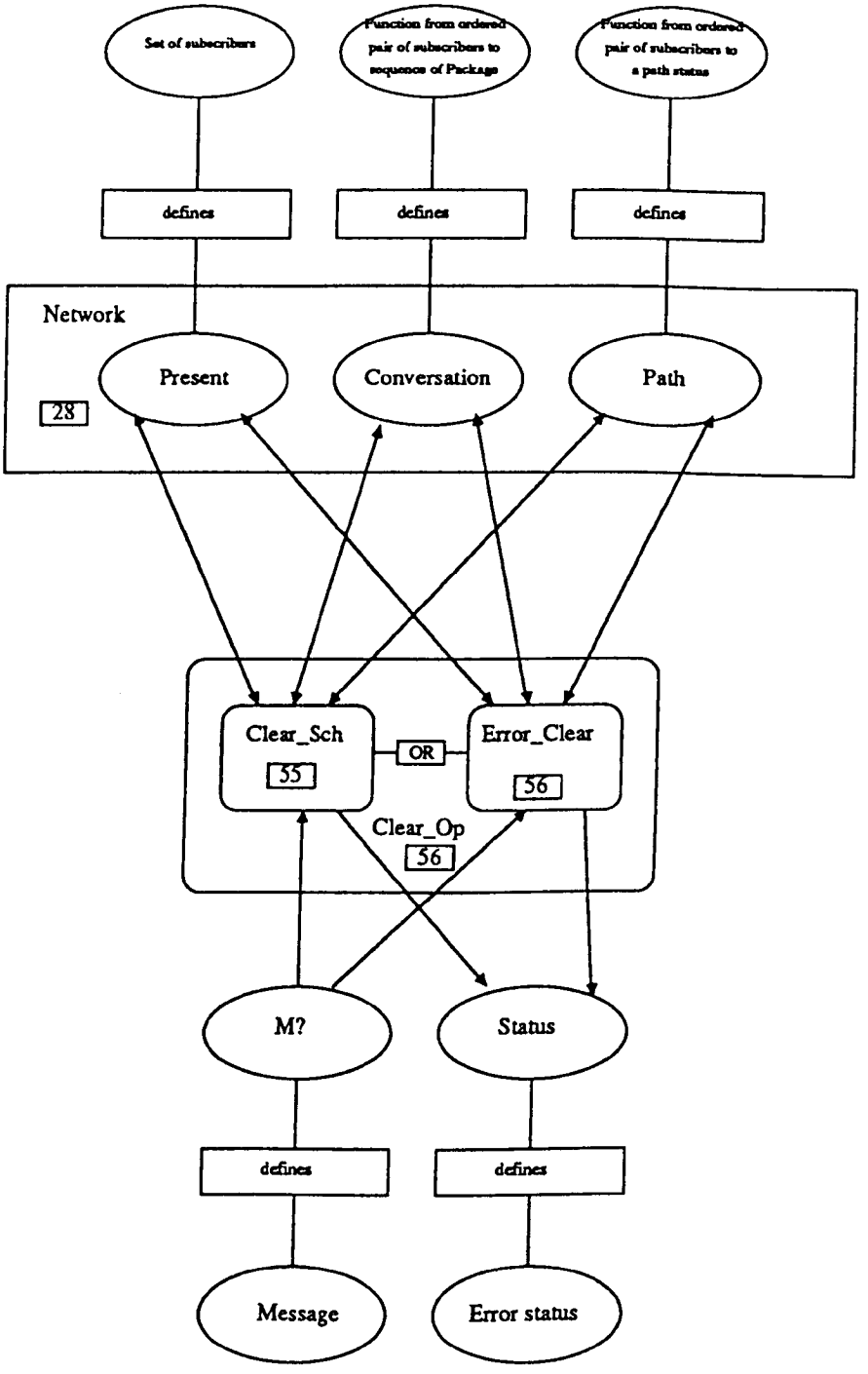
$$\Xi \text{Network}$$
$$M? : \text{Message}$$
$$\text{Status}' : \text{Error\_Status}$$

$$\exists \text{calling}, \text{called} : \text{SUB}; \text{op} : \text{Operation} \mid$$
$$M? = \text{setup}((\text{calling}, \text{called}), \text{op}) \land \text{op} = \text{clear} \land$$
$$(\text{Path}(\text{calling}, \text{called}) \neq \text{Established} \lor$$
$$\text{Path}(\text{called}, \text{calling}) \neq \text{Established} \lor$$
$$\text{Conversation}(\text{calling}, \text{called}) \neq \langle\rangle \lor$$
$$\text{Conversation}(\text{called}, \text{calling}) \neq \langle\rangle) \bullet \text{Status}' = \text{Error}$$

A robust version of the clear operation is defined in the schema Clear_Op defined below as the disjunction of the schemas Clear_Sch and Error_Clear.

$$\text{Clear\_Op} \triangleq \text{Clear\_Sch} \lor \text{Error\_Clear}$$

Figure 3.5 Interactions between the Clear Operation Schema and the State Schema



Proof Obligation for the Invariant of the Schema Clear_Op

CADiZ has ensured that the basic types are not violated.

The function *Path* is changed by the schema Clear_Sch such that elements of its domain are mapped to single elements in its range, therefore maintaining the type required by the functional override operator to ensure that the resulting type of *Path* is still a total function.

The schema Error_Clear does not change either the functions *Conversation* or *Path*, hence cannot violate their functional properties.

The preconditions of the schemas Clear_Sch and Error_Clear are mutually exclusive, hence there can be no conflict in changes of state.

Preconditions for Clear_Op

The preconditions for the clear operation are simplified to those represented by the schema Pre_Clear_Op_Simple defined below.

Pre_Clear_Op_Simple

$$\begin{array}{l} \text{Network} \\ M? : \text{Message} \\ \hline \exists\, calling, called : SUB \bullet M? = setup((calling, called), clear) \end{array}$$

The equivalence between the derived and actual preconditions are expressed the schema below.

$$Simplified\_3\_8 \triangleq pre\ Clear\_Op \Leftrightarrow Pre\_Clear\_Op\_Simple$$

3.3.5 The Send Operation

Data can be transmitted between a pair of subscribers after a call path is established between the subscribers.

The schema Send_Sch specifies the operation of sending data to another subscriber.

Send_Sch

ΔNetwork

M? : Message

Status' : Error_Status

$\exists$ sending, receiving : SUB; data : Package; op : Operation |

M? = active (((sending, receiving), data), op) $\wedge$ op = send $\wedge$

Path (sending, receiving) = Established •

Conversation' =

Conversation $\oplus$

{(sending, receiving) $\mapsto$

Conversation (sending, receiving) $\frown$ ⟨data⟩} $\wedge$ Status' = Okay

No change in the set Present and function Path

Present' = Present

Path' = Path

The data are represented by the type Package and assigned to the bound variable *data*. The input message *M?* has the components *sending*, *receiving*, *data* and *op* which are existentially quantified in the predicate part of the schema. The preconditions of the schema are that the operation, identified by *op*, is a send operation, and there is a path established between the *sending* and *receiving* subscribers. The postconditions are that the new data are concatenated on to the sequence of existing data for the conversation between the two subscribers and the value of the variable *Status'* is given the value of Okay.

Improving the Robustness

The schema Error_Send below has the preconditions required to catch any errors in the application of the send operation.

Error_Send

$$\Xi Network$$
$$M? : Message$$
$$Status' : Error\_Status$$

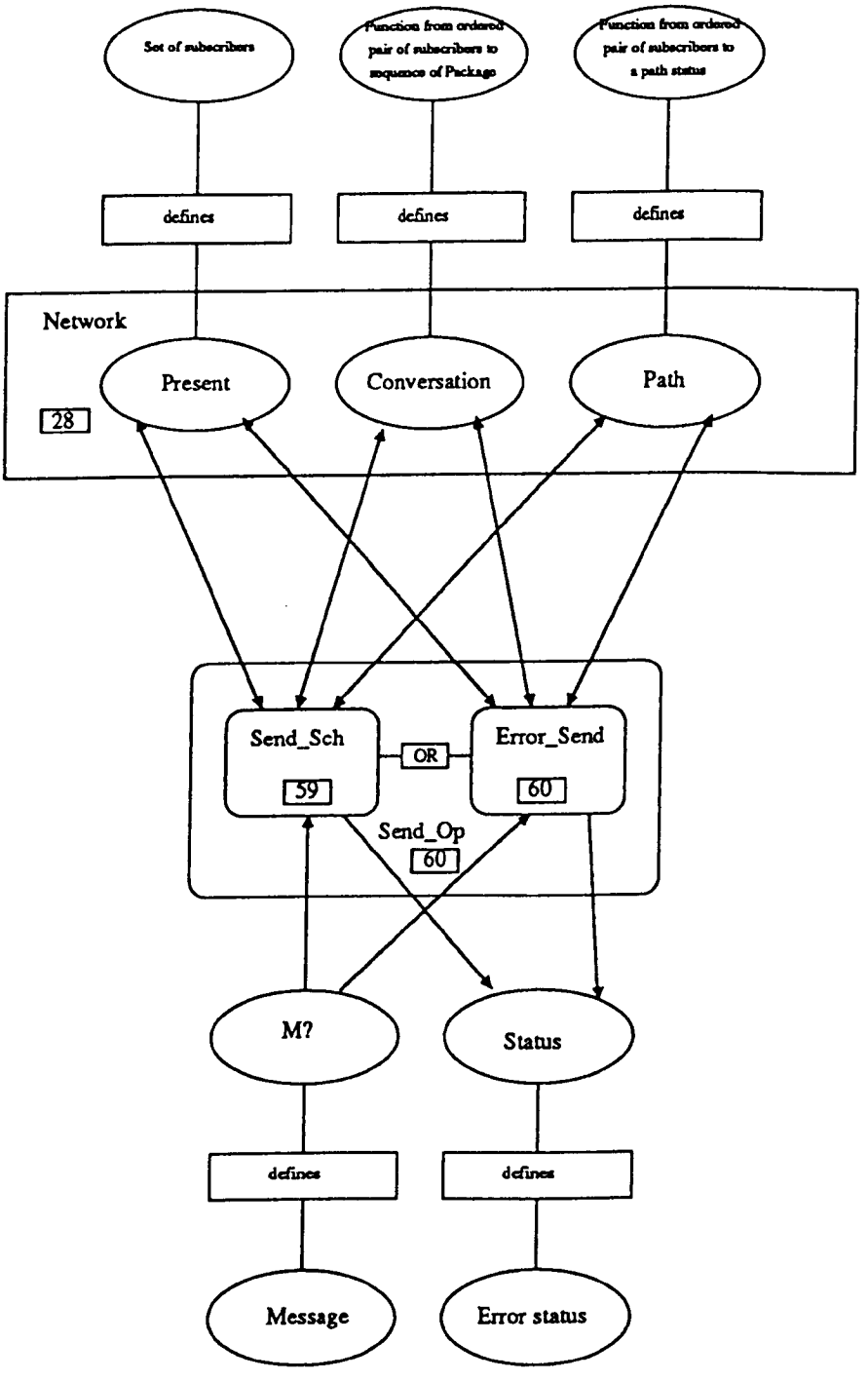
$$\exists\, sending, receiving : SUB;\ data : Package;\ op : Operation \mid$$
$$M? = active(((sending, receiving), data), op) \wedge op = send \wedge$$
$$Path(sending, receiving) \neq Established \bullet Status' = Error$$

A robust version of the send operation is defined as the schema Send_Op below.

$$Send\_Op \triangleq Send\_Sch \vee Error\_Send$$

Figure 3.6 shows the interactions between the schema Send_Op and the other components in the system. These interactions are considered in the following proof obligation.

Figure 3.6 Interactions between the Send Operation Schema and the State Schema



Proof Obligation for the Invariant of the Schema Send_Op

CADiZ ensures that the basic type rules are applied correctly.

The schema Send_Sch changes the function *Conversation* using the functional override operator such that the overring function is a single maplet, hence following the functional override type requirements to ensure that the resulting type of *Conversation* is a total function.

The schema Error_Send does not change either the functions *Conversation* or *Path*, hence cannot violate their functional properties.

The preconditions of the schema Send_Sch and Error_Send are mutually exclusive, hence cannot conflict about any changes of state.

Preconditions for Send_Op

The preconditions of the schema Send_Op are simplified to give the schema Pre_Send_Op_Simple below.

Pre_Send_Op_Simple

Network

M? : Message

$\exists$ sending, receiving : SUB; data : Package •

M? = active (((sending, receiving), data), send)

The correctness of the preconditions is expressed by the following schema definition.

Simplified_3_9 $\widehat{=}$ pre Send_Op $\Leftrightarrow$ Pre_Send_Op_Simple

The data transmitted to a subscriber must be accepted by a receive operation.

3.3.6 The Receive Operation

The schema Receive_Sch1 below specifies the effects of the receiving subscriber accepting data from the sending subscriber.

Receive_Sch1

ΔNetwork

M! : Message

Status' : Error_Status

$\exists$ sending, receiving : SUB; data : Package; op : Operation |

M! = active (((sending, receiving), data), op) $\wedge$ op = receive $\wedge$

Path (sending, receiving) = Established $\wedge$

Conversation (sending, receiving) $\neq \langle\rangle$ $\wedge$

data = head (Conversation (sending, receiving)) •

Conversation' =

Conversation $\oplus$

{(sending, receiving) $\mapsto$

tail (Conversation (sending, receiving))} $\wedge$ Status' = Okay

No change in the other Network declarations

Present' = Present

Path' = Path

The schema Receive_Sch1 uses the convention of representing output variables by identifiers ending with an exclamation mark, such as *M!* defined above. The preconditions for this schema are that a path is established between the sending and receiving subscribers, and the operation is of the type receive as indicated by the bound variable *op*. The preconditions do not involve any input variable. After the operation, the function *Conversation* is updated so that the tuple representing the sending and receiving subscribers maps to the tail of the value of the previous mapping. The value of the head of its previous value is bound

to the output variable and the status value is updated to be Okay.

The preconditions are expressed entirely in terms of state variables, hence an output variable can be assigned a value whenever the state of the system satisfies the preconditions. The preconditions of this schema can be satisfied at the same time as the preconditions of other schemas are satisfied. In general, such concurrent changes in a state are possible, but are contrary to a simple finite state machine model in which changes of state are caused by input events. In this case, because the values of the after state variables are defined uniquely, the operation specified by the schema Receive_Sch1 cannot be performed simultaneously with other operations and there is no possibility of it interfering with the changes of state defined by other schemas.

Alternative Schema for the Receive Operation

It is intuitively simpler to define the receive operation as an input operation and the data part of the input variable not being used by the schema. This leads to the physical interpretation that the subscriber generates an input when ready to receive a message. This interpretation is embodied in the schema Receive_Sch below.

Receive_Sch

ΔNetwork

M? : Message

M! : Message

Status' : Error_Status

$\exists$ sending, receiving : SUB; data, d : Package; op : Operation |

M? = active (((sending, receiving), d), op) $\wedge$

M! = active (((sending, receiving), data), op) $\wedge$ op = receive $\wedge$

Path (sending, receiving) = Established $\wedge$

Conversation (sending, receiving) $\neq \langle\rangle$ $\wedge$

data = head (Conversation (sending, receiving)) •

Conversation' =

Conversation $\oplus$

{(sending, receiving) $\mapsto$

tail (Conversation (sending, receiving))} $\wedge$ Status' = Okay

No change to the set Present and function Path

Present' = Present

Path' = Path

The input operation is included in the preconditions of the schema Receive_Sch and specifies the subscribers and the type of operation.

Improving the Robustness

The schema Error_Receive specifies the preconditions necessary to cater for error in the application of the receive operation. In this case the output takes the data value from the input operation because it cannot be obtained from the function *Conversation*.

Error_Receive

ΞNetwork
M? : Message
M! : Message
Status' : Error_Status

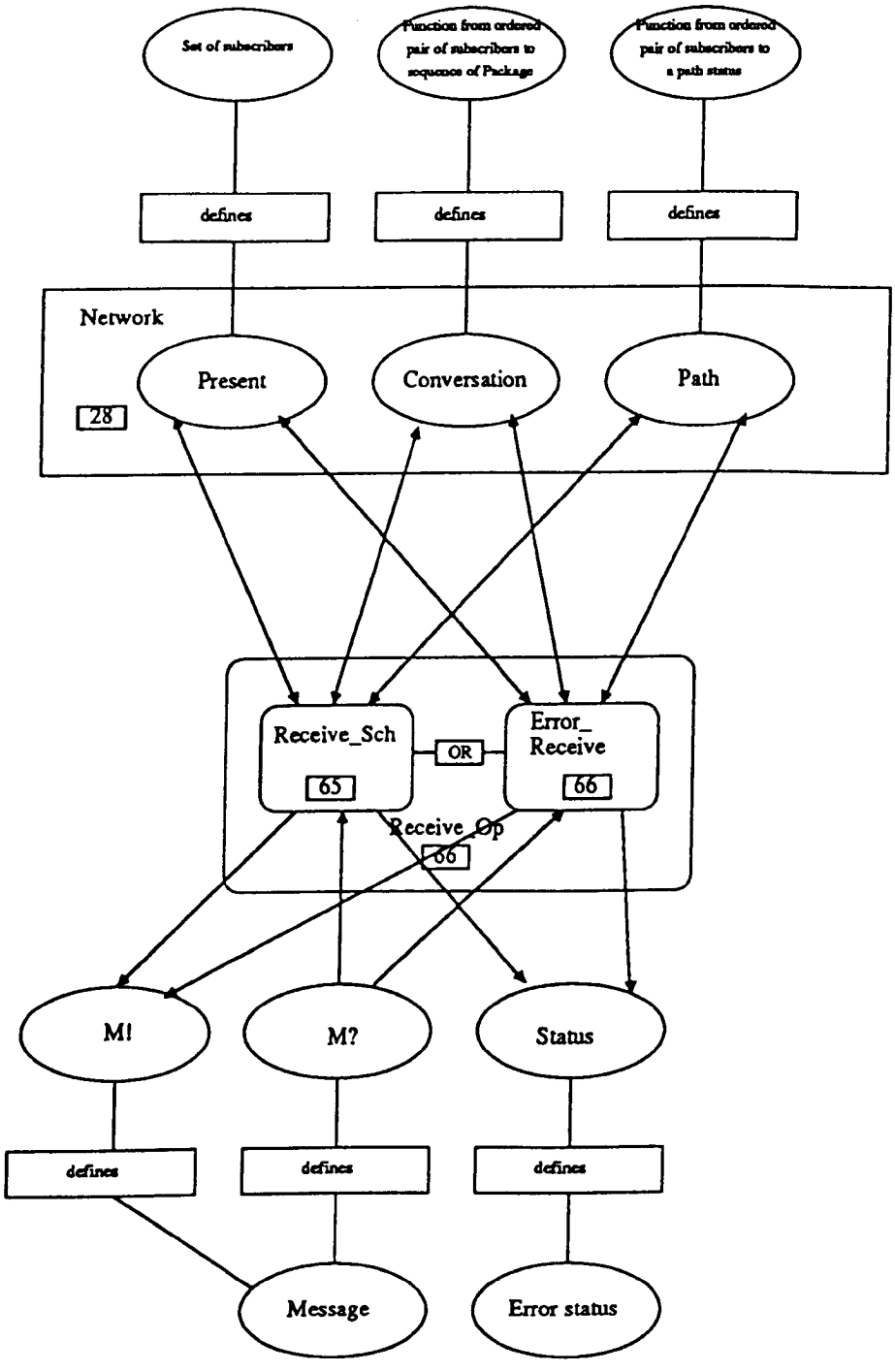
$\exists$ sending, receiving : SUB; d : Package; op : Operation |
M? = active (((sending, receiving), d), op) $\wedge$
M! = active (((sending, receiving), d), op) $\wedge$ op = receive $\wedge$
(Path (sending, receiving) $\neq$ Established $\vee$
Conversation (sending, receiving) = $\langle\rangle$) • Status' = Error

A robust version of the schema specifying the effects of the receive operation is defined as the schema Receive_Op below.

$$\text{Receive\_Op} \triangleq \text{Receive\_Sch} \vee \text{Error\_Receive}$$

Figure 3.7 shows the interactions between the schema Receive_Op and the other components in the system. These interactions are considered in the following proof obligation.

Figure 3.7 Interactions between the Receive Operation Schema and the State Schema



Proof Obligation for the Invariant of the Schema Receive_Op

CADiZ ensures that the basic type rules are applied correctly.

The schema Receive_Sch changes the function *Conversation* using the functional override operator such that the overriding function is a single maplet, hence following the functional override type requirements to ensure that the resulting type of *Conversation* is a total function.

The schema Error_Receive does not change either the functions *Conversation* or *Path*, hence cannot violate their functional properties.

The preconditions of the schema Receive_Sch and Error_Receive are mutually exclusive, hence cannot conflict about any changes of state.

Preconditions for Receive_Op

The preconditions of the schema Receive_Op are simplified to the schema Pre_Receive_Op_Simple below.

Pre_Receive_Op_Simple

Network

M? : Message

$\exists$ sending, receiving : SUB; data : Package •

M? = active (((sending, receiving), data), receive)

The equivalence between the simplified preconditions and rudimentary preconditions is represented by the schema Simplified_3_10 below.

Simplified_3_10 ≙ pre Receive_Op ⟺ Pre_Receive_Op_Simple

3.3.7 Network Implementation at this Design Stage

The effects of any operation performed on the system is specified to be the disjunction of the robust versions of the six operations.

Network_Imp ≙

Join_Op ∨ Leave_Op ∨ Call_Op ∨ Clear_Op ∨ Send_Op ∨ Receive_Op

The preconditions for this overall schema are the disjunction of the six preconditions for the operations, which covers all possible values of input messages, hence provides a total implementation of the communications network.

3.4 Formal Statement of the Safety Properties

Each of the properties stated informally in Section 3.1.2 is represented in this section as a predicate compatible with the schema Network_Imp. The mapping from informal to formal descriptions cannot be verified; other predicates can represent different valid interpretations of the informal descriptions. Each property is given as the right hand sequent (the consequent) of the theorem and the left hand sequent (the antecedent) contains the schema Network_Imp.

The specification of the safety properties is in the form of the five theorems 3.1 to 3.5 stated below.

Theorem 3.1 Conversation Definition

Properties 1: each conversation has two subscribers.

Network_Imp ⊢

$\exists S : \mathbb{P}(\mathbb{P}(SUB \times SUB)) \bullet dom\ Path \in S$

$\forall s1, s2 : SUB;\ p1, p2 : Path\_Status \bullet$

$Path(s1, s2) = p1 \land Path(s1, s2) = p2 \Rightarrow p1 = p2$

The above theorem states that the domain of *Path* has the type of the set of tuples of pairs of subscribers and that *Path* is a function. This theorem does not preclude a subscriber having more than one conversation established at the same time, but each conversation must have a separate pair of paths. The purpose of this theorem is to prevent the possibility of a path involving more that two subscribers, e.g. a tuple of the type (SUB × SUB × SUB) being an element of the domain of *Path*.

Proof of Theorem 3.1

Since the data type of *Path* is

$(SUB \times SUB) \longrightarrow Path\_Status$

the domain is given by

$dom\ Path = \mathbb{P}(SUB \times SUB)$

which is the require type.

Each operation schema has been verified to maintain the functional properties of *Path*, hence completing the proof.

Theorem 3.2 Privacy

Property 2: it is impossible for a third subscriber to receive data destined for the second subscriber of a conversation.

$$
\begin{aligned}
&\text{Network\_Imp} \vdash \\
&\quad \exists S : \mathbb{P}(\mathbb{P}(SUB \times SUB)) \bullet \text{dom Conversation} \in S \\
&\quad \forall s1, s2 : SUB;\ p1, p2 : \text{seq Package} \bullet \\
&\qquad \text{Conversation}(s1, s2) = p1 \wedge \\
&\qquad \text{Conversation}(s1, s2) = p2 \Rightarrow p1 = p2
\end{aligned}
$$

Proof of Theorem 3.2

The proof is identical to that for theorem 3.1, except that the function *Conversation* is specified instead of *Path*.

Theorem 3.3 Busy Subscriber

Property 3: subscribers can be busy.

$$
\begin{aligned}
&\text{Network\_Imp} \vdash \\
&\quad \exists m : \text{Message};\ calling, called : SUB;\ op : \text{Operation} \mid \\
&\qquad m = \text{setup}((calling, called), op) \wedge op = \text{call} \bullet \\
&\qquad\quad \text{Path}(called, calling) = \text{Established}
\end{aligned}
$$

Proof of Theorem 3.3

Providing the given set SUB has at least two non equal members, then it is possible to create the function *Path* such that there is a pair of non equal subscribers that maps to the identifier Established.

It follows that there is an implicit assumption about the given type SUB such that

$\# SUB > 1$

Thus, a subscriber can never be in an established conversation in a communications network that contains only one subscriber.

Theorem 3.4 Inaccessible Subscribers

Property 4: subscribers can be inaccessible.

$$\begin{aligned}&\text{Network\_Imp} \vdash \\ &\quad \exists\, m : \text{Message};\ \text{calling}, \text{called} : \text{SUB};\ \text{op} : \text{Operation} \mid \\ &\qquad m = \text{setup}((\text{calling}, \text{called}), \text{op}) \land \text{op} = \text{call} \bullet \text{calling} \notin \text{Present}\end{aligned}$$

Proof of Theorem 3.4

The proof is similar to that for property 3.

Theorem 3.5 No Duplicate Data

Property 5: data are received at most once.

Before stating the final form of the theorem, it is worthwhile to consider an initial attempt as defining this property.

$$
\begin{aligned}
&\text{Network\_Imp} \vdash \\
&\quad \forall \text{ data} : \text{Package}; \text{ sending, receiving} : \text{SUB} \bullet \\
&\qquad \# \\
&\qquad \{ m1 : \text{Message} \mid \\
&\qquad\ m1 = \text{active}(((\text{sending, receiving}), \text{data}), \text{receive}) \bullet m1 \} \leqslant \\
&\qquad \# \\
&\qquad \{ m2 : \text{Message} \mid \\
&\qquad\ m2 = \text{active}(((\text{sending, receiving}), \text{data}), \text{send}) \bullet m2 \}
\end{aligned}
$$

The intended interpretation of the above theorem is that the number of times any particular data value is received by a subscriber does not exceed the number of times the same value is sent. The theorem includes the possibility that the same data value being sent more than once between the same pair of subscribers. The messages are assumed in the above theorem to be identifiable uniquely by some means not visible at this level of abstraction, hence being separate messages.

However, it is not as simple as this because there is no history of operations attached to the schemas as they stand, nor is there any intrinsic sense of operations being executed in the Z notation as if it were a programming language executed by a computer.

Another attempt at specifying Property 5 involves defining an additional schema that is conjoined with the schema Network_Imp. The following schema Input_History records all the operations accepted by the system.

Input_History

$$\text{Input\_Trace} : \text{seq Message}$$

The component *Input_Trace* can be considered to be constructed by observing all messages that give rise to the component *Status'* being bound to the value *Okay*. This interpretation is represented by the schema Input_Monitor below that is conjoined with the schema

Network_Imp.

Input_Monitor

$$\Delta Input\_History$$
$$M? : Message$$
$$Status' : Error\_Status$$

$$Status' = Okay \land Input\_Trace' = Input\_Trace \frown \langle M? \rangle$$
$$\lor$$
$$Status' = Error \land Input\_Trace' = Input\_Trace$$

The schema Input_Monitor is total for both possible values of the component *Status'*, but the component *Input_Trace* is changed only in the cases of successful operations. The initial state for the schema Input_Monitor is given by the schema Initial_Input_Monitor

Initial_Input_Monitor

$$Input\_History'$$

$$Input\_Trace' = \langle\rangle$$

Proof of the Initial State

The empty sequence is obviously a valid binding of the sequence *Input_Trace.*

$$
\begin{array}{l}
\text{Network\_Imp} \wedge \text{Input\_Monitor} \vdash \\
\quad \forall\, m1, m2 : \text{Message};\ \text{sending}, \text{receiving} : \text{SUB}; \\
\quad\quad \text{data} : \text{Package} \mid \\
\quad\quad m1 = \text{active}\,(((\text{sending}, \text{receiving}), \text{data}), \text{receive}) \wedge \\
\quad\quad m2 = \text{active}\,(((\text{sending}, \text{receiving}), \text{data}), \text{send}) \bullet \\
\quad\quad\quad \text{count\_seq}\,(m1, \text{Input\_Trace}) \leqslant \text{count\_seq}\,(m2, \text{Input\_Trace}) \\
\text{where}
\end{array}
$$

$$
\text{count\_seq} : (\text{Message} \times \text{seq Message}) \longrightarrow \mathbb{N}
$$

$$
\begin{array}{l}
\forall\, m : \text{Message};\ h : \text{seq Message} \bullet \\
\quad (h = \langle\rangle \Rightarrow \text{count\_seq}\,(m, h) = 0) \vee \\
\quad (h \neq \langle\rangle \wedge m = \text{head}\, h \Rightarrow \\
\quad\quad \text{count\_seq}\,(m, h) = \text{count\_seq}\,(m, \text{tail}\, h) + 1) \vee \\
\quad (h \neq \langle\rangle \wedge m \neq \text{head}\, h \Rightarrow \\
\quad\quad \text{count\_seq}\,(m, h) = \text{count\_seq}\,(m, \text{tail}\, h))
\end{array}
$$

The above theorem defines a function called *count_seq* that evaluates the number of times a message occurs in a sequence of messages.

Proof of Theorem 3.5

The proof of this theorem can be seen by noting that from the schema Input_Monitor, each message of the form *m2* for a send operation in the component *Input_Trace* gives rise to the component *Status'* having the value *Okay*. In addition, from the schema Send_Op it follows that the sequence given by *Conversation(sending, receiving)* is changed by the value *data* added to it once.

Similarly for messages of the type *m1*, where the schema Receive_Op specifies that for each successful receive operation the value *data* is removed from the sequence

Conversation(sending, receiving) once. Therefore, because the receive process can never remove more elements than are sent, the number of messages of the type *m1* in the sequence *Input_Trace* can never exceed the number of occurrences of the messages of type *m2*.

Theorem 3.5 more accurately represents the intended interpretation of property 5 than the initial attempt. However, it still uses an informal interpretation of schemas being executed, but this time the past behaviour of schemas is recorded formally.

3.5 Additional Properties as a Consequence of the Implementation

The implementation described in the previous section results in some extra properties being included in subsequent stages in the design process as a consequence of the method of implementation. This section contains statements of theorems that represent these properties. The additional properties identified in the implementation are given an interpretation in the context of a communications network. If the additional properties generated by the implementation are not acceptable, then changes will have to be made to the schemas.

Import yorkkit

The standard tool kit does not contain a definition for functional inverse and the tool kit *yorkkit* is imported into this specification to provide the required definitions.

Theorem 3.6 Send Data During a Conversation

Property *A:* subscribers can only send data after a conversation has been established.

This is expressed as the following theorem:

Network_Imp ⊢

∀ m : Message; sending, receiving : SUB; op : Operation;

data : Package |

m = active (((sending, receiving), data), op) ∧ op = send ∧

Status' = Okay • Path (sending, receiving) = Established

Proof of Theorem 3.6

This theorem follows from the specification of the send operation given by the schema Send_Sch, in which the function *Path* must denote an established path for the sending and receiving pair of subscribers.

Theorem 3.7 Clear after all Data has been Sent

Property *B:* a conversation is cleared after all data has been received.

The following theorem specifies Property *B:*

Network_Imp ⊢

∀ m : Message; calling, called : SUB; op : Operation |

m = setup ((calling, called), op) ∧ op = clear ∧

Status' = Okay •

Conversation (calling, called) = ⟨⟩ ∧

Conversation (called, calling) = ⟨⟩

Proof of Theorem 3.7

The specification of the clear operation given in the schema Clear_Sch is used to verify the above theorem. The predicate in the schema Clear_Sch ensures that the function *Conversation* must denote empty sequences for the calling and called subscribers in both orders for the *Status'* to have the value *Okay*.

Theorem 3.8 Data Received in Order Sent

Property *C:* data are received in the order they are sent.

The following theorem contains the predicate for Property *C:*

$$
\begin{aligned}
&\text{Network\_Imp} \wedge \text{Input\_Monitor} \vdash \\
&\quad \forall\, s1, s2, r1, r2 : \text{Message};\ \text{sending}, \text{receiving} : \text{SUB}; \\
&\qquad d1, d2 : \text{Package} \mid \\
&\qquad s1 = \text{active}\,(((\text{sending}, \text{receiving}), d1), \text{send}) \wedge \\
&\qquad s2 = \text{active}\,(((\text{sending}, \text{receiving}), d2), \text{send}) \wedge \\
&\qquad r1 = \text{active}\,(((\text{sending}, \text{receiving}), d1), \text{send}) \wedge \\
&\qquad r2 = \text{active}\,(((\text{sending}, \text{receiving}), d2), \text{send}) \bullet \\
&\qquad\quad \text{Input\_Trace}^{-1}\, s1 < \text{Input\_Trace}^{-1}\, s2 \Rightarrow \\
&\qquad\qquad \text{Input\_Trace}^{-1}\, r1 < \text{Input\_Trace}^{-1}\, r2
\end{aligned}
$$

The predicate in this theorem states that if a data packet, *d1*, is sent before another data packet, *d2*, from one subscriber to another, then the first packet must be received before the second packet. The order in which data packets are sent and received is given by their positions in the sequence *Input_Trace*.

Proof of Theorem 3.8

The above theorem is proved from the properties of sequences in the two schemas Send_Sch and Receive_Sch.

The schema Send_Sch adds new data to the sequence given by *Conversation(sending, receiving)* by concatenating the value to the end of the existing sequence. Similarly, the schema Receive_Sch obtains data by removing the first element in the sequence given by *Conversation(sending, receiving)*. Therefore, the order in which elements are stored and removed from the sequence maintains a first-in, first-out queue discipline, which ensures that the order in which elements are removed is the same as in which the elements are stored.

Theorem 3.9 One Path between Two Subscribers

Property *D:* there can only be one communications path between each pair of subscribers.

This is defined by the predicate in the theorem below:

$$
\begin{aligned}
&\text{Network\_Imp} \vdash \\
&\quad \forall \text{ calling, called} : \text{SUB} \bullet \\
&\qquad \# \\
&\qquad \{ s : \text{Path\_Status} \mid ((\text{calling, called}), s) \in \text{Path} \bullet \\
&\qquad\quad (\text{calling, called}) \} = 1
\end{aligned}
$$

Proof of Theorem 3.9

This theorem is verified from the properties of functions. Since *Path* is defined to be a function, and none of the operation schemas violate the functional properties of *Path*, each pair of subscribers in its domain maps to a single element in its range.

Whether these extra properties are continued throughout subsequent stages in a design

process depends on how the subsequent implementations are verified. For example, if the Z schemas represent an implementation of the system at the next stage in the design process and the schemas introduced in this chapter are verified to be equivalent:

$Network\_Imp \Leftrightarrow Network\_Next\_Imp$

then the extra properties will be retained.

However, if the original properties are used to verify:

$Network\_Next\_Imp \Rightarrow Property\_Set$

then different extra properties can result.

3.6 Formal Specification of the Liveness Properties

The safety properties specify the functional behaviour in terms of *only good things will happen*. The liveness properties specify that *something will happen*. The network implemented in this chapter is represented by the schema Network_Imp, which defines each change of state in response to an input signal. The occurrence of input signals is not within the control of the implementation. Any liveness specification must be dependent on the external activities in the form of input signals.

Temporal logic has been used to specify the liveness properties with Z schemas [Duke89], however, introducing temporal logic necessitates a new notation and deviates from the predicate logic defined in the semantics of standard Z notation.

The approach taken here is similar to that by Duke et al. [Duke88] and defines sequences of all schema states and messages that are experienced by the system. Such a sequence can be infinite and, obviously, does not actually exist because it looks into the future. The sequences of schema states is a convenient notion to specify the type of dynamic behaviour required of a system. This approach is very similar to that of using the schema Input_History in Section 3.4, where the schema Input_History records all the input messages received by the system.

The state of the network model is given by the schema Network and a history of states therefore has the type of a sequence of elements of the type Network, similarly a history of

messages has the type of a sequence of elements of the type Message.

History

State_Trace : seq Network

Input_Trace : seq Message

State_Trace 1 ∈ {Initial_State}

The schema History specifies the sequences *State_Trace* and *Input_Trace*. This schema does not give any other information about the sequences, except that the first state must be a valid initial state. The sequences are for specification purposes only and there is no suggestion that such sequences are implemented. However, it is useful to have the interpretation of some monitor that records all changes of state that accompany messages that give rise to the error status *Okay*. This interpretation is represented by the schema State_Monitor below.

State_Monitor

ΔHistory

ΔNetwork

M? : Message

Status' : Error_Status

Status' = Okay $\wedge$ Input_Trace' = Input_Trace $\frown$ ⟨M?⟩ $\wedge$

State_Trace' = State_Trace $\frown$ ⟨θ Network'⟩

$\vee$

Status' = Error $\wedge$ Input_Trace' = Input_Trace $\wedge$

State_Trace' = State_Trace

The schema State_Monitor is total for both possible values of *Status'* and the after state value of Network is not constrained by this schema beyond its signature. This leaves the changes of state to be determined by the network implementation schema Network_Imp.

The initial state for the schema State_Monitor is given by the schema Initial_State_Monitor.

Initial_State_Monitor

History'

Input_Trace' = ⟨⟩

State_Trace' 1 $\in$ {Initial_State}

Proof of the Initial State

The empty sequence is a valid binding of the sequence *Input_Trace* and Initial_State

represents valid bindings of the schema Network. Therefore, the initial value of *State_Trace* also meets the invariants of the schema History.

From the definition of the initial state, it follows that the after state value that arises from an input message in the *ith* position in the sequence *Input_Trace* will appear in the *(i + 1)th* position in the sequence *State_Trace*. This fact is used to express the theorems in this section.

The proof sketches are based on the existence of the sequences *Input_Trace* and *State_Trace*, hence they presuppose that state changes occur in response to messages. The main objective of this section is to indicate how liveness properties can be specified in Z in relation to an implementation. The proof sketches do not set out to verify that the implementation has the required liveness properties. This cannot be done because the implementation is at a too high level of abstraction. What the proof sketches set out to verify is that the state changes that occur are described by the appropriate operation schema. The existence of the traces themselves must be proved at a lower level of abstraction that deals with the required level of detail to guarantee the response to changes. This means that the proofs are conditional on an interpretation of any future implementation of the system.

One of the basic liveness properties is that: *providing the preconditions are satisfied, an input causes a change of state*. This section illustrates how theorems can express this liveness property for each of the network operations.

The specification of the liveness properties take the form of theorems 3.10 to 3.15 given below.

The format for each consequence of the theorems for all the operations is

$$\textit{input message} \land \textit{before state meets preconditions} \Rightarrow \textit{after state meets preconditions}$$

where the input message is given by *Input_Trace*, and the before and after states are given by *State_Trace*.

Theorem 3.10 Join Operation

Informally this theorem states that operations requesting a subscriber to join the network

such that the subscriber is not already a member, then the operations will be successful.

$$
\begin{aligned}
&\text{Network\_Imp} \wedge \text{State\_Monitor} \vdash \\
&\quad \forall i : \mathbb{N};\ sub : SUB;\ m : Message;\ state, state' : Network \bullet \\
&\qquad \text{State\_Trace } i = state \wedge \text{Input\_Trace } i = m \wedge \\
&\qquad m = member(sub, join) \wedge \text{State\_Trace } (i + 1) = state' \wedge \\
&\qquad sub \notin state.Present \Rightarrow sub \in state'.Present
\end{aligned}
$$

Proof of Theorem 3.10

From schema Join_Sch the preconditions for a change of state are that the operation represented by the input *M?* is a join operation and the subscriber is not a member of the set *Present*. For the predicate in the schema Join_Sch to be true, the component *Present* is given by:

$Present' = Present \cup \{sub\}$

hence

$sub \in Present'$

making the term

$sub \in state'.Present$

in the theorem true.

Theorem 3.11 Leave Operation

For all leave operations successfully performed in the network, there are two states such that in one state the subscriber was a member of the network and in the next state the same subscriber was not a member.

$$
\begin{aligned}
&\text{Network\_Imp} \land \text{State\_Monitor} \vdash \\
&\quad \forall i : \mathbb{N};\ sub : SUB;\ m : Message;\ state, state' : Network \bullet \\
&\qquad \text{State\_Trace}\ i = state \land \text{Input\_Trace}\ i = m \land \\
&\qquad m = member(sub, leave) \land \text{State\_Trace}\ (i + 1) = state' \land \\
&\qquad (\forall u : SUB \bullet \\
&\qquad\quad \text{Conversation}\ (u, sub) = \langle\rangle \land \\
&\qquad\quad \text{Conversation}\ (sub, u) = \langle\rangle \land \text{Path}\ (u, sub) = Free \land \\
&\qquad\quad \text{Path}\ (sub, u) = Free) \land sub \in state.Present \Rightarrow \\
&\qquad\quad sub \notin state'.Present
\end{aligned}
$$

Proof of Theorem 3.11

The schema Leave_Sch includes the preconditions of the operation represented by the input *M?* is a leave operation, the subscriber, *sub*, is a member of the set *Present*, the function *Conversation* maps all pairs in its domain that includes *sub* to the value of an empty sequence and the function *Path* similarly maps all such pairs of subscribers to the value *Free*. This corresponds to the left hand predicate of the implication in the consequence of the theorem. The after state given in the schema Leave_Sch includes the term

$$Present' = Present \setminus \{sub\}$$

Therefore,

$$sub \notin Present'$$

hence, the right hand predicate in the implication is true.

Theorem 3.12 Call Operation

For all call operations such that the preconditions of the operations are satisfied, then there must be a change of state as specified by the schemas.

$$
\begin{aligned}
&\text{Network\_Imp} \land \text{State\_Monitor} \vdash \\
&\quad \forall\, i : \mathbb{N};\ \text{calling}, \text{called} : \text{SUB};\ m : \text{Message}; \\
&\qquad \text{state}, \text{state}' : \text{Network} \bullet \\
&\qquad\quad \text{State\_Trace}\ i = \text{state} \land \text{Input\_Trace}\ i = m \land \\
&\qquad\quad m = \text{setup}\ ((\text{calling}, \text{called}), \text{call}) \land \\
&\qquad\quad \text{State\_Trace}\ (i + 1) = \text{state}' \land \text{calling} \in \text{state}.\text{Present} \land \\
&\qquad\quad \text{called} \in \text{state}.\text{Present} \land \text{calling} \neq \text{called} \land \\
&\qquad\quad \text{state}.\text{Conversation}\ (\text{calling}, \text{called}) = \langle\rangle \land \\
&\qquad\quad \text{state}.\text{Conversation}\ (\text{called}, \text{calling}) = \langle\rangle \land \\
&\qquad\quad \text{state}.\text{Path}\ (\text{calling}, \text{called}) = \text{Free} \land \\
&\qquad\quad \text{state}.\text{Path}\ (\text{called}, \text{calling}) = \text{Free} \Rightarrow \\
&\qquad\qquad \text{state}'.\text{Path}\ (\text{calling}, \text{called}) = \text{Established} \land \\
&\qquad\qquad \text{state}'.\text{Path}\ (\text{called}, \text{calling}) = \text{Established}
\end{aligned}
$$

Proof of Theorem 3.12

The left hand predicate of the implication in the theorem corresponds to the preconditions of the schema Call_Sch. The postconditions of the schema Call_Sch include the term

$$Path' = Path \oplus \{(calling, called) \mapsto Established\} \oplus \{(called, calling) \mapsto Established\}$$

Therefore,

$$Path'(calling, called) = Established \land Path'(called, calling) = Established$$

and the right hand predicate of the implication in the theorem is true.

Theorem 3.13 Clear Operation

Similarly for the clear operation.

Network_Imp ∧ State_Monitor ⊢

∀ i : ℕ; calling, called : SUB; m : Message;

state, state' : Network •

State_Trace i = state ∧ Input_Trace i = m ∧

m = setup ((calling, called), clear) ∧

State_Trace (i + 1) = state' ∧

state . Path (calling, called) = Established ∧

state . Path (called, calling) = Established ∧

state . Conversation (calling, called) = ⟨⟩ ∧

state . Conversation (called, calling) = ⟨⟩ ⇒

state' . Path (calling, called) = Free ∧

state' . Path (called, calling) = Free

Proof of Theorem 3.13

The left hand predicate of the implication in the theorem corresponds to the preconditions of the schema Clear_Sch . The postconditions of the schema Clear_Sch include the term

$$Path' = Path \oplus \{(calling, called) \mapsto Free\} \oplus \{(called, calling) \mapsto Free\}$$

Therefore,

$$Path'(calling, called) = Free \wedge Path'(called, calling) = Free$$

and the right hand predicate of the implication in the theorem is true.

Theorem 3.14 Send Operation

For all send operations that meet the preconditions, the next state of the system is such that the data included in the operation are the last element of the sequence of data mapped by the two subscribers in the conversation.

Network_Imp ∧ State_Monitor ⊢

∀ i : ℕ; sending, receiving : SUB; data : Package;

m : Message; state, state' : Network •

State_Trace i = state ∧ Input_Trace i = m ∧

m = active (((sending, receiving), data), send) ∧

State_Trace (i + 1) = state' ∧

state . Path (sending, receiving) = Established ⇒

last (state' . Conversation (sending, receiving)) = data

Proof of Theorem 3.14

The left hand predicate of the implication in the consequence of the theorem corresponds to the preconditions of the schema Send_Sch . The predicate in the schema Send_Sch includes the term

$$Conversation' = Conversation \oplus \{(sending, receiving) \mapsto Conversation\,(sending, receiving) \frown \langle data \rangle\}$$

hence

$$last\,(Conversation'\,(sending, receiving)) = data$$

Thus, denoting that the right hand predicate of the implication in the consequence of the theorem is true.

Theorem 3.15 Receive Operation

In the case of all successful receive operations, the element at the head of the sequence of data between two subscribers is removed.

Network_Imp ∧ State_Monitor ⊢

∀ i : ℕ; sending, receiving : SUB; data, d : Package;

m : Message; state, state' : Network •

State_Trace i = state ∧ Input_Trace i = m ∧

m = active (((sending, receiving), d), receive) ∧

State_Trace (i + 1) = state' ∧

state . Path (sending, receiving) = Established ∧

state . Conversation (sending, receiving) ≠ ⟨⟩ ⇒

state' . Conversation (sending, receiving) =

tail (state . Conversation (sending, receiving))

Proof of Theorem 3.15

The left hand predicate of the implication in the consequence of the theorem corresponds to the preconditions of the schema Receive_Sch. The predicate in the schema Receive_Sch includes the term

$$Conversation' = Conversation \oplus \{(sending, receiving) \mapsto tail\,(Conversation\,(sending, receiving))\}$$

hence

$$Conversation'\,(sending, receiving) = tail\,(Conversation\,(sending, receiving))$$

Thus, denoting that the right hand predicate of the implication in the consequence of the theorem is true.

3.7 Discussion

This section contains some comments on the approaches taken with the Z notation in this chapter, with particular emphasis on combining schemas to form complete descriptions of the implementation.

3.7.1 Verification Conditions

The goal of verification is to prove that the implementation represented in the Z notation does exhibit the required properties of the system. The type of proof that is required depends on the design approach.

The strongest relationship between a specification and an implementation is an equivalence relation:

$$Specification \Leftrightarrow Implementation$$

The equivalence relation means that whenever the specification is true so is the implementation, and whenever the implementation is true so is the specification. For the equivalence relation to be applicable, the specification and implementation must use the same state space. The correctness of the equivalence relation means that the specification and implementation are true in the same states and define the same state transitions.

A weaker relation between a specification and an implementation is an implication relations:

$$Implementation \Rightarrow Specification$$

Whenever the implementation is true it follows that the specification must also be true. However, the converse is not necessarily true and the specification can be true in conditions in which the implementation is false. For the implication relation to be used in the above form, the implementation must be a subset of the state space of the specification. The interpretation of an implementation in the implication relation is that the implementation is more restrictive (stronger) that the specification. However, this may not always be the case and implementations may be required to be less restrictive (weaker) than specifications.

This means that the implication relation is in the opposite direction:

$Specification \Rightarrow Implementation$

The type of implication relation that is appropriate will depend on the type of behaviour that is being compared. In the case of static behaviour, where the implementation is required to be applicable in more states than the specification (less restrictive), the implication relation is:

$Specification \Rightarrow Implementation$

Since it is only the before states and input variables that are relevant, the relation is better expressed as:

$pre\ Specification \Rightarrow pre\ Implementation$

In the case of dynamic behaviour, where the implementation is more restrictive than the specification, the implication relation is:

$Implementation \Rightarrow Specification$

In the cases where the specification is given in terms of properties, the implication relation is:

$Implementation \Rightarrow Property$

If the implementation uses a different state space to the specification, there must be a homomorphic relation, ϕ, between the two state spaces such that:

$Implementation \Rightarrow \phi\ (Specification)$

where

$\phi : abstract\_states \leftrightarrow concrete\_states$

Once again, the direction of the implication relation depends on the type of behaviour that is being compared. Because of the different forms the verification condition can take, it is

necessary to explain what aspects of the implementation and specification are being considered so that the verification condition is interpreted correctly.

3.7.2 Isolated Operation Specification

For comparison between complete descriptions for a design process and isolated descriptions for specification only, this section contains two examples of schemas written as specifications of operations in isolation of each other. The schemas Join_Sch_Loose and Send_Sch_Loose below are based on the schemas Join_Sch and Send_Sch.

Join_Sch_Loose

$$\Delta Network$$
$$Sub? : SUB$$
$$Status' : Error\_Status$$

$$Sub? \notin Present$$
$$Conversation' =$$
$$Conversation \oplus \{u : SUB \bullet (u, Sub?) \mapsto \langle\rangle\} \oplus$$
$$\{u : SUB \bullet (Sub?, u) \mapsto \langle\rangle\}$$
$$Path' =$$
$$Path \oplus \{u : SUB \bullet (u, Sub?) \mapsto Free\} \oplus$$
$$\{u : SUB \bullet (Sub?, u) \mapsto Free\}$$
$$Present' = Present \cup \{Sub?\}$$
$$Status' = Okay$$

The schema Join_Sch_Loose does not include any information about how the schema determines that it is a join operation. In the schema Join_Sch, the type of operation is included in the input message variable.

Send_Sch_Loose

ΔNetwork

Sub1?, Sub2? : SUB

Data? : Package

Status' : Error_Status

Path (Sub1?, Sub2?) = Established

Conversation' =

Conversation $\oplus$

$\{(Sub1?, Sub2?) \mapsto Conversation\ (Sub1?, Sub2?)\ ^\frown \langle Data? \rangle\}$

Status' = Okay

Present' = Present

Path' = Path

The input variables in the schema Send_Sch_Loose do not identify the operation to be performed and there is no association between the different input variables in the schema to indicate how the data are supplied to the system.

The isolated schema terms such as Join_Sch_Loose and Send_Sch_Loose can be connected by logical operators, but the resultant schema cannot be used to identify which operation causes the change of state, see Section 2.3.

3.7.3 Disjunction of Schemas

The disjunction of the six operation schemas forming the network specification schema has the effect of merging all the declarations. Those variables with the same name must have the same signature to avoid clashes. The composite predicate is the disjunction of the separate predicates in the six operation schemas. In this study, the operation schemas are chosen to give a total specification for all possible input messages for a particular operation and each operation schema excludes the other five from their preconditions. This can be

proved as a theorem of the implementation as a desirable property of the implementation.

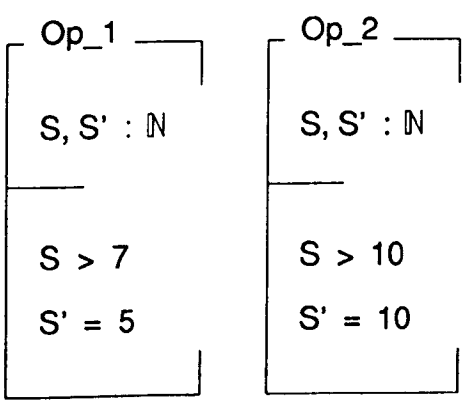
If the schema Receive_Sch1 is used instead of Receive_Sch, then the input message does not form any part of the preconditions of that schema and all possible input messages are included in the other five operation schemas. The network is specified as:

$$Network\_Imp1 \mathrel{\widehat{=}} Join\_Op \lor Leave\_Op \lor Call\_Op \lor Clear\_Op \lor Send\_Op \lor Receive\_Sch1$$

Which is again a disjunction of six operation schemas, but this time their preconditions are not disjoint.

The introduction of the schema Receive_Sch1 raises questions about sharing variables between schemas and, although the Z notation is not a programming language, raises questions about possible interference between schemas that are not mutually exclusive. Such interference is excluded by making the schemas specify the after state values uniquely.

The non determinism that can occur when the preconditions are satisfied for more than one operation is illustrated by the following simple schema that is formed using the disjunction of two non exclusive schemas Op_1 and Op_2.



$$Op\_1\_or\_Op\_2 \mathrel{\widehat{=}} Op\_1 \lor Op\_2$$

Expanding the definition of the schema Op_1_or_Op_2 gives the schema below.

Op_1_or_Op_2_Expand

$$S, S' : \mathbb{N}$$

$$S > 7 \land S' = 5$$
$$\lor$$
$$S > 10 \land S' = 10$$

The value of S' when S is greater than 10 is non deterministic. Either Op_1 or Op_2 can have effect, but the only way to determine which one is to examine the after state.

The effects of the disjunction of schemas should be checked both for semantical sense in the interpretation of the model and for syntactical sense for the whole of the system, i.e. that it is acceptable or desirable to have such non determinism in the context of the required behaviour of the system.

3.7.4 Style of Writing Schemas

All the schemas presented in this chapter are written in the style:

$\exists$ *declarations* | *preconditions* • *postconditions*

This makes it a simple act to identify the terms that affect different aspects in the predicate. The actual preconditions are still verified from the preconditions of the schemas by simplification.

The equivalence relation between the rudimentary preconditions given by the *pre* operator and the simplified preconditions is expressed as a schema equation so that CADiZ can be used to expand the schema definitions.

3.7.5 Concurrent Activities

Activities that occur simultaneously are called concurrent. If the activities have access to the same information (e.g. share data), then the activities can interfere and cause unwanted results. The ability of activities to occur concurrently both reduces the time required to complete the activities and removes the need to ensure that activities do not occur simultaneously, hence concurrency is useful.

Questions about concurrent activities of systems specified in the Z notation can be avoided by assuming that all operations are performed instantaneously and no two operations can be performed at the same instant. In physically realizable implementations of systems, instantaneous performance of operations is not possible and it is necessary to consider the effects of operations being performed concurrently.

Appendix B.4 contains a discussion of representing concurrent activities in the Z notation.

3.8 Summary

This chapter contains a study of using the Z notation both to specify some basic properties of a communications network and to describe an implementation of a communications network. Proof sketches are given to verify that the implementation implies the properties.

The implementation presented in this chapter represents the first stage of a design process, hence it is useful to include details of how the schemas are combined to represent the whole system and still retain all the information about the operations.

The interactions between schemas are given a diagrammatic representation in this chapter to highlight the effects caused by particular operation schemas. The action of drawing the diagrams is also very useful for uncovering mistakes in the schemas. The mistakes come to light because the schemas provide the information for drawing the diagrams, hence must be studied carefully.

3.8.1 Complete Description of an Implementation

An implementation of a communications network is described by a set of Z schemas. These schemas are combined to give a single schema, Network_Imp, that represents the complete description of the implementation. Each operation is described separately, but is an integrated manner based on an input component common to all operations. The properties of the schema Network_Imp are analysed formally in Sections 3.4 to Section 3.6. This analysis is simplified by the type of operation being readily identified with an input message, thereby selecting parts of the schema Network_Imp that are relevant to the operation associated with the input message.

3.8.2 Proof Obligations

The schema equations must be checked to ensure that the schemas are combined in sensible ways. Apart from the data refinement rules, there are no general guide lines as to what are the checks. The proof obligations discharged in this chapter are:

1 The given sets and free type definitions are consistent.

2 The invariants are maintained by all the operation schemas.

3 At least one valid state exists in the form of an initial state.

4 The preconditions are calculated and simplified. The preconditions of all the operations are compared to ensure that all input conditions are covered and no ambiguity is introduced.

5 The implementation is verified with respect to formal statements of the required properties.

All the verifications are presented in the form of proof sketches. Proof sketches, although not as formal as is possible with the Z notation, do give a strong indication of the correctness of statements about schema equations. Any problems or difficulties encountered in developing a proof sketch will also arise in constructing proof demonstrations and, whereas proof demonstrations are very difficult to complete, proof sketches are relatively easy to construct.

The proof sketches in this chapter have been very elementary due to simple data definitions and operation schemas used in the study, however, they are still needed to reduce the possibility of making mistakes and to reveal assumptions about the implementation. For instance, the proof sketches in Section 3.4 resulted in the requirement for the size of the given set SUB to be greater than one. The type rules for the Z notation is enforced by the use of CADiZ, thereby simplifies the requirements of the proof sketches.

3.8.3 Preconditions

The preconditions for each operation schema in this chapter are expanded and simplified to express the conditions under which the operation can occur. Examples of the expansion of preconditions are given in Section 3.3 and an example of the simplifications is given in Appendix B.2.

The preconditions in this study show that:

1 The implementation is defined under all conditions. Since each operation schema is defined for all conditions for that operation and the collection of operations includes all possible input values, the complete implementation is defined under all condition.

2 Each operation is disjoint from the other operations. Because the input value identifies explicitly a single operation schema, no two operation schemas can respond to the same input value and have disjoint behaviours.

3.8.4 Postconditions

The after states of all the operation schemas are specified in a constructive style such that the after state variables incorporate the before state variables and change the binding only to allow the one operation to take place. This style is very straightforward to use when considering each operation in isolation from other events.

The double ended arrows in the interaction diagrams indicate that all the operation schemas use both the before and after versions of the state components. A non constructive style of defining schemas is discussed in the context of concurrency in Appendix B.4, where state changes can occur as a result of multiple operations occurring simultaneously.

3.8.5 Properties

One of the main reported advantages of using a formal notation for the initial stage specification of a system is that subsequent implementations can be compared and verified against this specification, hence increasing the confidence in the design.

One of the important points stressed in this chapter is the importance of unambiguous specification of the required properties to prevent different interpretations being applied to the same property. Although initially the properties are stated informally in Section 3.1.2, they are given a formal representation as predicates that are compatible with the Z schemas in Sections 3.4 and 3.6. It is this formal representation that expresses the properties throughout the design process. However, it must be remembered that there is no method of verifying that the formal descriptions of the properties represent the informally stated requirements correctly.

3.8.6 Liveness

A concept of possible histories, or traces, of messages is introduced in this chapter to state theorems about some of the properties required of the system. Since there are no intentions of actually implementing such variables, the histories of messages do not form part of the implementation, instead they are used to represent properties required of the implementation expressed at the current level of abstraction. Eventually these properties will have to be exhibited by an implementation at a lower level of abstraction.

The liveness behaviour of the implementation is specified in this chapter by the device of analysing a sequence of possible state values (or bindings) of the schema Network. This data structure has strong analogies with state transition diagrams, but uses a more abstract and flexible representation.

Chapter 4
Replicated Database Systems

This chapter is an introduction to replicated database systems that provides a context for the descriptions in the Z notation contained in Chapters 5 and 6. This introduction includes descriptions of a general model of a database system and some of the concurrency control problems that arise in replicated database systems.

4.1 Introduction

A database is a collection of data objects, where each data object has a value which is either read or changed by writing a new value. A database system is both the hardware and software that supports access to the database accessed by several users simultaneously. That is, data is shared by the users. The sharing of data gives rise to both physical and organizational problems. The physical problems are a consequence of limitations of hardware that cannot support simultaneous access of the same components. The organizational problems are a consequence of the need for maintaining consistency between the data objects during changes activated by different users. Some of the organizational problems are addressed in this chapter.

An ideal replicated database system contains replicas of the data objects stored at distributed sites. Physical constraints add to the problem of maintaining consistency between data ob-

jects that are nominally identical. Any change to one copy of a data object must be reflected in all copies.

The model of database systems described in this chapter is based on the work of Bernstein, Hadzilcos and Goodman [Bern87], and is an abstraction of the many different types of database management systems, transaction processing systems and file systems.

The access of the data held in the database system is modelled as a transaction, where a transaction contains a number of **read** and **write** operations performed on data objects. Each operation is viewed as an atomic event, so a read or write operation cannot be broken down into more basic operations. A transaction is terminated by either a **commit** operation, to effect any changes to the data objects, or an **abort** operation, to cancel any changes to the data objects referred to in the transaction. A data object appears at most once for a read and a write operation within the same transaction, i.e. no multiple read operations or write operations involve the same data object. Some additional restrictions are sometimes imposed, such as a read operation on a data object must always appear before a write operation on the same data object within transactions [Abbad89].

Database systems impose restrictions on the execution of transactions to ensure that each transaction accesses shared data without interfering with other transactions. This is known as *concurrency control.*

The particular problem of database systems addressed in this chapter is concurrency control. The types of problems that can arise with concurrency control are appreciated by considering an example of the way in which operations in transactions can interfere. A banking system updates the amount of money held by customers. The amount held by a customer is updated by a transaction that reads the current balance, adds the value of the new deposit and writes the new value. This type of transaction can be invoked by any user. Consider two transactions, where transaction number 1 deposits £50 and transaction number 2 deposits £25 to the same account. Assuming that transactions are executed simultaneously, the following sequence of events is possible:

1 transaction number 1 reads the current balance of £150

2 transaction number 2 reads the current balance of £150

3 transaction number 1 writes the new balance of £200

4 transaction number 1 commits its operations

5 transaction number 2 writes the new balance of £175

6 transaction number 2 commits its operations.

The deposit of £50 by transaction number 1 is lost.

This is an example of the way transactions can interfere and arises due to the interleaving of operations contained in different transactions.

One method of ensuring that transactions do not interfere is to prevent the operations in different transactions interleaving. This is known as *serial operation*. Serial operation prohibits database systems from executing transactions concurrently, this makes very inefficient use of its resources.

Database systems can allow some interleaving of operations in different transactions if the effects are equivalent to the same transactions executed in some serial order. This is known as *serializable operation*. As any serial operation is correct, it follows that the equivalent serializable operation is also correct because the results are the same.

The previous example of the two deposits made to the same bank account is not serializable because there is no equivalent serial operation that produces the same effect. That is, if either the serial order of transaction number 1 followed by transaction number 2, or transaction number 2 followed by transaction number 1 was executed, the value of the final balance would be £225, not £175.

The main reasons for using replicated databases are:

1 To increase availability. The database system can still operate even if some sites have failed.

2 To improve performance. Holding data at geographically close sites can improve the speed of access to the data objects.

One of the apparently contradictory objectives of a replicated database system is that it should behave like a one copy database, apart from availability and performance. A correct-

ness criterion of replicated databases is called *one copy serializable* [Bern87], which means that the execution of the transactions has the same effects as the equivalent transactions being executed in some serial order on a one copy database. If a replicated database system produces the same effects as a one copy database system, then it is considered correct. A replicated database system with the one copy serialization property both exhibits the correct behaviour of a one copy database system, and has better availability and performance properties than a one copy database system.

4.2 Serializability Theory

The concept of conflicting transactions is used in concurrency control, where two operations in different transactions conflict if they both operate on the same physical data object and at least one of the operations is a write.

The execution of the operations in a group of transactions is known as a *history* and it is this history of operations that is referred to in the serializability theory.

4.2.1 Serialization Graphs

The serializability theory presented by Bernstein, Hadzilcos and Goodman is based on a diagram called a *serialization graph* [Bern87]. A serialization graph is a directed graph whose nodes are the committed transactions in a history and has edges *(Ti, Tj)*, where $i \neq j$, if an operation in transaction *Ti* precedes and conflicts with an operation in transaction *Tj*. A very similar notation to that described by Bernstein, et al., is used below. The main difference is that a more restricted interpretation of histories of the transactions is used so that the history gives the exact order of all the operations, not just the essential ones referred to in the history diagrams in the book *Concurrency Control and Recovery in Database Systems* [Bern87]. That is, the histories in this section give the total order of operations, instead of a partial order.

Serialization graphs are drawn for *complete histories*. The term a 'complete history' means a sequence of operations such that all the transactions are either committed or aborted. That is, no proper subsets of operations in a transaction are included in the history.

In the notation used for transactions, data objects are identified by the lower case letters, such as x and y. For replicated data objects it is necessary to distinguish between logical ob-

jects and physical objects. A logical object refers to all the copies of the same data object, whereas a physical object refers to a particular copy of that object. To distinguish between the physical and logical objects, letters, such as *a* and *b*, identify the site holding a copy of the data object. Physical data objects are identified by two letter abbreviations, such as *xa* and *yb*.

The operations that are contained in transactions are **read, write, commit** and **abort**. These are abbreviated as *r*, *w*, *c* and *a* respectively. Transactions are numbered to allow operations to be identified with a transaction, hence, *r1* is a read operation in transaction number 1. Bringing all the terminology together, a read operation in transaction 1 on the physical data object *xa* is abbreviated as *r1(xa)*.

Examples below are similar to those in the book *Concurrency Control and Recovery in Database Systems* [Bern87] and illustrate the application of the serializability theory.

Consider the history:

$$H1 = r1(xa)\ r2(xa)\ w1(xa)\ r3(xa)\ w2(yb)\ c2\ w1(yb)\ w3(xa)\ c1\ c3$$

which is a consequence of executing the following three transactions concurrently:

$$T1 = r1(xa)\ w1(xa)\ w1(yb)\ c1$$

$$T2 = r2(xa)\ w2(yb)\ c2$$

$$T3 = r3(xa)\ w3(xa)\ c3$$

The conflicting operations in the history *H* are:

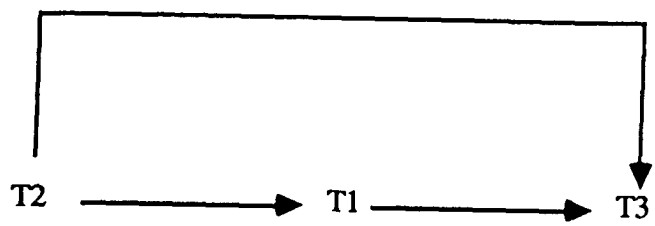
$T1 \rightarrow T3$ for data object *xa*

$T2 \rightarrow T1$ for data objects *xa* and *yb*

$T2 \rightarrow T3$ for data object *xa*

The serialization graph is shown in Figure 4.1.

Figure 4.1 Serialization Graph for H1



The serializability theorem states that a history, *H*, is serializable if and only if the serialization graph of *H* is acyclic [Bern87].

In the above example the serialization graph is acyclic and an equivalent serial order of the transactions is: *T2 T1 T3*

which has the serial history of:

r2(xa) w2(yb) c2 r1(xa) w1(xa) w1(yb) c1 r3(xa) w3(xa) c3

An example of a non serializable history of the same transactions is:

H2 = r1(xa) r2(xa) w1(xa) w1(yb) r3(xa) w2(yb) c1 w3(xa) c2 c3

The transactions that are in conflict are:

T1 → *T3* for data object *xa*

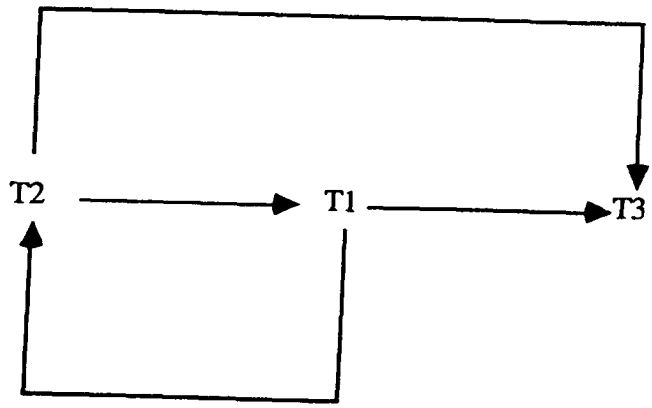
T2 → *T1* for data object *xa*

T1 → *T2* for data object *yb*

T2 → *T3* for data object *xa*.

The serialization graph is given in Figure 4.2.

Figure 4.2 Serialization Graph for H2



The serialization graph of history *H2* has the cycle *T2 T1 T2*, which means that there is no equivalent serial order of transactions. Neither *T2* before *T1*, nor *T1* before *T2* has the same effects as the history *H2*. This can be seen by observing that the operation *r2(xa)* appears in *H2* before *w1(xa)*, implying that *T2* must appear before *T1* in a serial order to have the same effect. However, the operation *w1(yb)* appears before *w2(yb)* in history *H2*, implying that *T1* must appear before *T2* in a serial history, hence there is no equivalent serial history of *H2*.

4.2.2 Serialization Graphs for Replicated Databases

The criterion of one copy serializability requires that all the histories of the transactions performed on a replicated database system are equivalent to a serial order of the equivalent transactions performed on a database system that does not have replicated data objects [Bern83, Bern87]. In the case of one copy database systems, there is only one physical data object for each logical data object.

Consider the history of the transactions performed on a replicated database system:

H3 = w1(xa) w1(xb) w1(yc) w1(yd) c1 r2(xa) w2(yc) c2 r3(yd) w3(xb) c3

of the transactions:

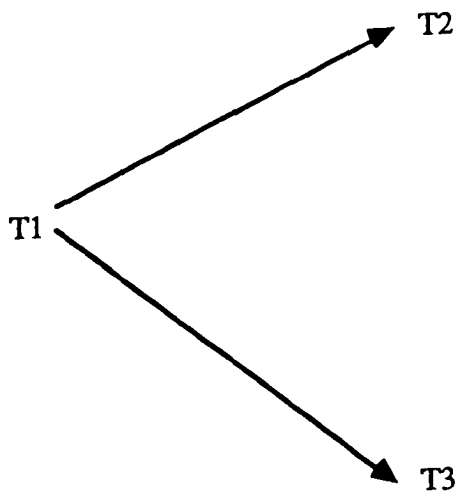
T1 = w1(xa) w1(xb) w1(yc) w1(yd) c1

T2 = r2(xa) w2(yc) c2

T3 = r3(yd) w3(xb) c3

The history *H3* is a serial history of the transactions *T1 T2 T3*, hence is serializable and has the serialization graph shown in Figure 4.3.

Figure 4.3 Serialization Graph for H3



However, when considering the one copy serializability the logical data objects have to be accessed. The transactions become:

T1' = w1(x) w1(y) c1

T2' = r2(x) w2(y) c2

T3' = r3(y) w3(x) c3

The equivalent versions of the data objects in the replicated database system must cause the transactions to appear in the following order:

1 transactions *T2'* and *T3'* have to be executed after transaction *T1'* as both read the latest version of data object *x*

2 transaction *T2'* has to be executed before *T3'* as *T2'* uses the version of data object *x* before transaction *T3'* has updated it

3 *T3'* has to be executed before *T2'* as *T3'* uses the version of data object *y* before transaction *T2'* has updated it.

It follows that it is impossible to find an equivalent one copy serial history for *H3*.

4.2.3 Replicated Data Serialization Graphs

The problem of cyclic dependency is identified in a modified version of the serialization graph, called *replicated data serialization graph* [Bern83, Bern87].

A replicated data serialization graph of a history is the serialization graph for the same history, but with edges added such that the following two conditions hold for all logical data objects *x*:

1 if transaction *Ti* and *Tk* write to data object *x*, then either there is a path from *Ti* to *Tk* or there is a path from *Tk* to *Ti*

2 if the following cases apply

(a) transaction *Tj* reads from data object *x* the value that was written to *x* from transaction *Ti*

(b) transaction *Tk* writes some copy of data object *x*, where $k \neq i$ and $k \neq j$

(c) there is a path from transaction *Ti* to transaction *Tk*

then there is also a path from *Tj* to *Tk*, this is known as a *read before path.*

The read before path indicates that transaction *Tj* reads data object *x* logically before transaction *Tk* writes *x* [Bern83].

Condition 1 imposes a write order for *H* and condition 2 imposes a read order for *H.*

Using the second condition for the history *H3*:

1 transaction *T2* reads a copy of data object *x* from transaction *T1*

2 transaction *T3* writes a copy of data object *x*

3 there is a path from *T1* to *T3* (indicating that transaction *T1* precedes *T3*).

It follows that there is an added path between transaction *T2* and transaction *T3*, indicating that transaction *T2* precedes transaction *T3*.

Similarly,

1 transaction *T3* reads a copy of data object *y* from transaction *T1*

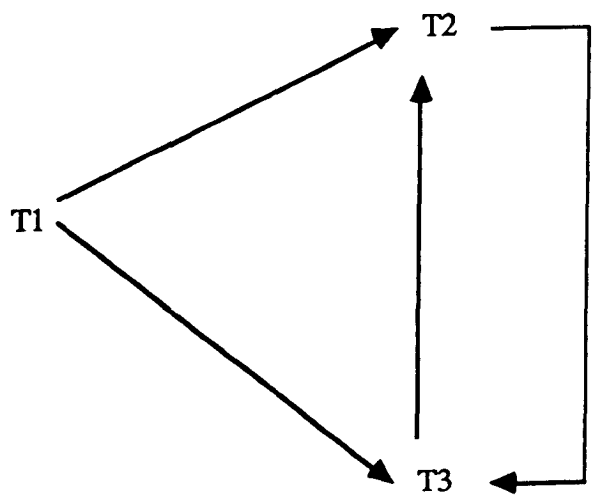
2 transaction *T2* writes a copy of data object *y*

3 there is a path from transaction *T1* to transaction *T2* (indicating that transaction *T1* precedes *T2*).

It follows that there is an added path between *T3* and *T2*, indicating that transaction *T3* precedes *T2*.

A replicated data serialization graph for history *H3* is shown in Figure 4.4. Note that there is not a unique replicated data serialization graph for this history, but cycles in the graphs are the same for all versions.

Figure 4.4 Replicated Data Serialization Graph for H3



The one copy serialization theorem in the book *Concurrency Control and Recovery in Database Systems* [Bern87] asserts that if a replicated data history, *H*, has an acyclic replicated data serialization graph, then *H* is one copy serializable.

Chapter 5 represents serialization graphs and replicated data serialization graphs in the Z notation as sets of end points. The expression of the serializability theory in the Z notation is used as the specification of the replicated database system implemented in Chapter 5.

4.3 Concurrency Control Techniques in Replicated Database Systems

This section contains descriptions of two techniques for concurrency control that are addressed in this thesis. For a review of the problem refer to the paper by Bernstein and Goodman [Bern81], which gives a survey of the techniques up to the time of writing that paper.

4.3.1 Two Phase Locking

A two phase locking technique prevents conflicts between concurrent operations by applying read locks and write locks.

Before executing a read or write operation on a data object, a transaction must establish ownership of that object by preventing, or locking out, all other transactions that contain operations that conflict with the operation to be performed by the transaction claiming ownership of the object.

The ownerships of the locks are determined by the two rules [Bern81]:

1 Different transactions cannot simultaneously own conflicting locks.

2 Once a transaction relinquishes a lock, it can never obtain an additional lock.

The second rule leads to the transaction obtaining the locks in two phases; a growing phase and a shrinking phase. This enforces a serializable order on the transactions. A proof is presented in the book by Bernstein, et al., [Bern87].

Multiple copies of data objects are handled by a read lock on a single copy of the data object and a write lock on all copies of the data object before a write operation is executed. This technique is known as *read once, write all.* If a lock cannot be granted because the data object is already allocated to a transaction executing a conflicting operation, the operation is delayed. This can lead to deadlock, with one transaction waiting for a second transaction to complete, however, the second transaction cannot complete because it is waiting for the first transaction to complete. Moreover, the read once, write all algorithm does have some problems in cases of network or site failures.

Variations of the basic two phase locking technique are described by Bernstein and Goodman [Bern81].

A read once, write all concurrency control procedure expressed in the Z notation is contained in Chapter 5.

4.3.2 Quorum Consensus

The concept of quorums is also used in quorum consensus algorithms, where a quorum is a set of sites that are accessed in the execution of an operation.

The read and write operations are performed on quorums of sites according to the following rules [Herl86]:

1 The initiating site sends an invocation request to the transaction manager, which forwards it to an initial quorum of database copies.

2 Each database in the initial quorum sends its log to the transaction manager.

3 The transaction manager merges the logs to construct a composite history called the view. The transaction manager determines the response from this view of the current state of the database system.

4 The transaction manager generates a new entry and appends the new entry to the composite history. This updated view is sent to all the databases in a final quorum for the operation.

5 When all the databases in the final quorum acknowledge the update, the transaction manager returns the response to the initiating site.

An operation is aborted if the transaction manager cannot complete the above actions.

In this technique of concurrency control, there must be a consensus between each read quorum and write quorum such that [Bern83]:

1 The intersection between each read and write quorum is non empty.

2 The intersection of each pair of write quorums is non empty.

The sizes of the read and write quorums can be assigned to optimise the performance of the replicated database system. Also, the read and write quorums can be changed dynamically to reflect changes in the database system due to communications links and site failures [Herl86].

A model expressed in the Z notation of a quorum consensus concurrency control algorithm is contained in Chapter 6.

Chapter 5
Serializability Constraint on Replicated Database Systems

This chapter applies the Z notation to the problem of concurrency control of replicated database systems. Chapter 4 contains an introduction to replicated database systems and the mathematical basis of the one copy serialization property. The study presented in this chapter uses the Z notation for both expressing a property oriented specification of a replicated database system and for modelling the concurrency control aspects of an implementation. In particular, the serialization property is written as a specification, and a simple two phase locking read once, write all protocol is implemented. Both the specification and the implementation are written in the Z notation. The main verification theorem is that all the possible histories of operations conforming to the description of the replicated data objects, also conform to the specification of the one copy serialization property. Although the theorem stating the verification theorem is expressed elegantly in the Z notation, the means of proving the theorem using both the specification and implementation is not immediately clear.

The main points that should be drawn from this chapter are:

1 The implementation is formed by combining several schema terms together such that not only are all the operations defined, but so is the method of invoking the operations.

2 Once an operation has been defined by a Z schema, the preconditions of the schema are calculated and simplified to ensure that the preconditions are sufficient for the required span of state space.

3 Proof sketches are useful both for discharging proof obligations of the Z notation and for verifying an implementation with respect to a specification.

The specification of the one copy serializability property is contained in Section 5.1.

An implementation of a replicated database system with a two phase locking method with a read once, write all algorithm is contained in Sections 5.2 to 5.5. The implementation represents the functional behaviour of the whole system in terms of the changes of state caused by the four database operations that can be performed (i.e. read, write, commit and abort) on data objects. All the schemas follow a unified approach to implementing a replicated database system such that the schemas, when combined, retain information about when the operations are invoked. That is, the applicability of each operation schema is determined by values of input variables and state variables, not informal text associated with the schemas.

Section 5.6 contains two proof sketches of the one copy serialization property. The first proof sketch proves by an induction argument that all histories of operations executed by the implementation must be one copy serializable. The second proof sketch shows that all possible sequences of operations possible by the implementation are also possible by the specification of the one copy serialization property, hence the implementation must also have the one copy serialization property.

Section 5.7 contains a summary of the main findings of this chapter.

Appendix C contains examples and discussions of topics relevant to the study of replicated database systems.

Diagrams are included in the informal descriptions of schemas to provide insight into the interaction between schemas and to help cross referencing between the main components of the descriptions in this Chapter.

All the schemas defined in both the specification and implementation of the study are included in this chapter. This is both for convenience of presentation and to illustrate the level of abstraction used in this study.

Table 5.1 lists the sections that contain the descriptions of the elements of the design stage given most emphasis in this chapter.

Table 5.1 Summary of Design Stage in Chapter 5

Section	Description
5.1	Formal description of the one copy serializability property in the Z notation in the form of the schema Serializable_History
5.2 - 5.5	Description in the Z notation of an implementation of a replicated database system in the form of the schema DBS_Imp
5.6	Proof sketches of the verification of implementation

5.1 One Copy Serializability Property

This section describes the construction of the schema Serializable_History that represents the one copy serialization property. The property is expressed as a set of all histories that exhibit the one copy serialization property. In addition to the restriction of the one copy serializability property, the order in which operations can occur in any history is restricted by a precedence relation that is represented by the set *Trans_Precedence*.

5.1.1 Data Definitions

Before specifying the one copy serialization property in a schema it is necessary to define the data types and operations that are assumed by this schema.

Import toolkit

Import yorkkit

The tool kits provided by CADiZ contain the definition of data types such as natural number and postfix operations such as inverse.

[LOGICAL_OBJECT, VALUE, LETTER]

The first given set in this chapter is LOGICAL_OBJECT which represents the data objects (or entities) stored in the database. They are called logical because they refer to all physical copies of the same data object. The second given set is VALUE which refers to the set of values that can be stored in the data objects. The third given set is LETTER which identifies each of the sites that contains a copy of the database.

Site == LETTER

The type Site is defined to be syntactically equivalent to the given set LETTER. This ties up with the notation used to represent data objects in Chapter 4.

$$Trans\_Num == \mathbb{N}$$

Each transaction is identified uniquely by a transaction number, which is represented by the type Trans_Num and has the type of a set of natural numbers.

$$Physical\_Object == (LOGICAL\_OBJECT \times Site)$$

The physical objects stored in the database system are represented by the type Physical_Object which has components for LOGICAL_OBJECT and Site.

Proof Obligation for the Consistency of the Given Sets

The given sets do not have any operations performed on them to change their values and are used for identification only. This means that the given sets introduced above are consistent because they can be replaced by standard types such as ASCII characters for LETTER, real numbers for VALUE and strings of alphanumeric characters for LOGICAL_OBJECT to give a consistent model.

$$Operator ::= read \mid write \mid commit \mid abort$$

The free definition of the type Operator defines the four identifiers that represent the four operators in transactions executed by a database system.

Proof Obligation for the Free Type Definition of Operator

All four branches of the definition are non recursive, hence the free type definition is consistent.

$$Op ::= access \ll ((((\{read, write\} \times Trans\_Num) \times Physical\_Object) \times VALUE) \gg \mid end \ll (\{commit, abort\} \times Trans\_Num) \gg$$

The operations executed by the database system are represented by the type Op which is defined as a free type with two branches representing the two types of operations.

Proof Obligation for the Free Definition of Op

Both branches of the definition are non recursive, hence the definition of Op is consistent.

The first branch is identified as access and represents the operations using either read or write operators. The second branch is identified as end and represents the operations using either commit or abort operators.

The access branch has elements composed of the components of either a read or a write operator, a Trans_Num element, a Physical_Object and a VALUE making a tuple with four elements. The end branch has components of the form of ordered pairs of elements, composed of either a commit or an abort operator and a Trans_Num element.

A record of all the operations executed by the database system is maintained in subsequent schemas. This record is represented by the data type of a sequence of operations.

The operations represented by Op are uniquely identifiable. The transaction numbers are unique and a single read and/or a single write operation is at most performed once to/from the same data object within a transaction. Since each operation in a history of operations is unique, the inverse function of the history will give the single position that the operation occurs in the sequence of operations executed by the database system.

5.1.2 Conflicting Transactions

Two operations conflict when they both refer to the same data object and at least one of the operations is a write operation.

$$Conflicting\_Operator == \{(read, write), (write, read), (write, write)\}$$

The type Conflicting_Operator is defined syntactically to be the set of ordered pairs of operators that can conflict.

The schema History_Invariant defines an invariant property of the historical record of all operations executed on the replicated database system.

History_Invariant

$$History\_Rec : seq\ Op$$
$$Trans\_Precedence : \mathbb{P}\ (Op \times Op)$$

The invariant is that the history record is consistent with the partial order of operations within transactions

$$\forall o1, o2 : Op \mid (o1, o2) \in Trans\_Precedence \bullet$$
$$History\_Rec^{-1}\ o1 < History\_Rec^{-1}\ o2$$

Note that sequences are equivalent to functions from integers to elements. This means that functional inverses of sequences give the positions in which elements occur. If elements in a sequence are unique, then the inverse of the sequence will give the single position in the sequence of an element.

The function *History_Rec* can be regarded as an input that must satisfy the conditions given in the predicate part of the schema. The schema History_Invariant restricts the possible functions for *History_Rec* to those that are consistent with the partial order of operations supplied by the set *Trans_Precedence*. The set *Trans_Precedence* has its members

predefined such that each member is an ordered pair of operations. The first operation contained in each pair must occur before the second in all histories of operations. The usual precedence relation defined in *Trans_Precedence* is that operations are executed in the order in which they occur in the transactions and that transactions can be executed in any order.

The schema Precede_Set below defines a set that indicates whether one operation occurs before another.

Precede_Set

$$\begin{array}{|l}
\text{History_Invariant} \\
\text{Precedes} : \mathbb{P}\,(\text{Op} \times \text{Op}) \\
\hline
\text{Precedes} = \\
\{o1, o2 : \text{Op} \mid \text{History_Rec}^{-1}\ o1 < \text{History_Rec}^{-1}\ o2 \bullet (o1, o2)\} \\
\end{array}$$

The schema Precede_Set defines the set *Precedes* to be the set of ordered pairs of operations such that the first element occurs before the second in the history of operations that is given by the function *History_Rec*.

The schema Conflicting_Set below defines the set of all transactions that conflict.

Conflicting_Set

$$
\begin{array}{l}
\text{Precede\_Set} \\
\text{Conflicting} : \mathbb{P}\ (\text{Trans\_Num} \times \text{Trans\_Num}) \\
\hline
\text{Conflicting} = \\
\{\text{num1}, \text{num2} : \text{Trans\_Num} \mid \\
\quad \exists\ \text{o1}, \text{o2} : \text{Op};\ \text{op1}, \text{op2} : \text{Operator}; \\
\qquad \text{obj1}, \text{obj2} : \text{Physical\_Object};\ \text{v1}, \text{v2} : \text{VALUE} \bullet \\
\qquad\quad \text{o1} = \text{access}\ (((\text{op1}, \text{num1}), \text{obj1}), \text{v1}) \land \\
\qquad\quad \text{o2} = \text{access}\ (((\text{op2}, \text{num2}), \text{obj2}), \text{v2}) \land \\
\qquad\quad (\text{o1}, \text{o2}) \in \text{Precedes} \land \text{obj1} = \text{obj2} \land \text{num1} \neq \text{num2} \land \\
\qquad\quad (\text{op1}, \text{op2}) \in \text{Conflicting\_Operator} \bullet (\text{num1}, \text{num2})\}
\end{array}
$$

The schema Conflicting_Set constructs the set of pairs of transaction numbers that refer to transactions containing operations that conflict in the history defined by the sequence *History_Rec* used in the schema History_Invariant.

The schema Conflicting_End_Points below defines the set of all transactions that conflict either directly or indirectly.

Conflicting_End_Points

Conflicting_Set

Conflicting_Points : ℙ (Trans_Num × Trans_Num)

Conflicting_Points =

Conflicting ∪

{n1, n2, n3 : Trans_Num |

(n1, n3) ∈ Conflicting_Points ∧ (n3, n2) ∈ Conflicting_Points •

(n1, n2)}

The schema Conflicting_End_Points gives a constructive definition of the set *Conflicting_Points* that contains all the ordered pairs of transaction numbers that refer to transactions containing operations that conflict in a sequence of transactions. The schema Conflicting_Set identifies the edges in the serialization graph for the given history. The schema Conflicting_End_Points as identifies all the end points of paths in the serialization graph.

The schema Conflicting_Trans pulls together the other schemas in this subsection to define the invariant for serializable history. The invariant is equivalent to the requirement that there are no cycles in the serialization graph of the history of operations.

Conflicting_Trans

Conflicting_End_Points

Invariant for the serializability property

∀ n : Trans_Num • (n, n) ∉ Conflicting_Points

Proof Sketch of the Representation of Serialization Graphs

The validity of the Z description depends on the schemas representing serialization graphs correctly.

The representation of serialization graphs can be proved correct by considering the construction of the set *Conflicting_Points* and comparing it with the construction of the serialization graph for new operations as they are added to the history record.

The constraints of the set *Trans_Precedence* applies to the values of the history record that is used to construct both the set *Conflicting_Points* and the serialization graph. The restriction on completed histories applies to both the set *Conflicting_Points* and the serialization graph once all the transactions have completed their operations.

When the sequence *History_Rec* is either empty or has one element there can be no conflicting operation, hence the serialization graph is either blank or a single node for the transaction number. The set *Precedes* is empty.

An induction argument can deal with more complicated sequences. Starting with the sequence *History_Rec* with two conflicting operations.

Let

$History = \langle opi, opj \rangle$

where *opi* is an operation in transaction *i* and *opj* is an operation in transaction *j*.

There are two cases to consider:

1 The operations are either in the same transactions or in different transactions and do not conflict.

   (i) The serialization graph will either consist of a single node if the operations are in the same transaction, or two isolated nodes if the operations are in different transactions.

(ii) The set *Precedes* will contain just one member

$$Precedes = \{(opi, opj)\}$$

From the schema Conflicting_Set, the set *Conflicting* will be empty because the predicate is not satisfied. Therefore, the set *Conflicting_Points* will also be empty.

2 The operations are in different transactions and the operations conflict.

(i) The serialization graph will consist of a single edge between two nodes.

(ii) The sets *Precedes*, *Conflicting* and *Conflicting_Points* will have a single member

$$Precedes = Conflicting = Conflicting\_Points = \{(i, j)\}$$

The induction step for the proof assumes that the history record has some arbitrary value and the set *Conflicting_Points* correctly represents the corresponding serialization graph. Let

$History\_Rec = h$

Assume that a new operation *opk* is added, representing an operation in transaction *k*.

There are two possibilities to consider.

1 Operation *opk* does not conflict with any operation in the history *h* that are in a different transaction.

(i) The serialization graph does not change.

(ii) The set *Precedes* has new members added to represent all previous operation preceding the new operation *opk*. The new members will be of the form

$$\forall\, opi : Op \mid opi \in ran\ h \bullet (opi, opk) \in Precedes$$

The set *Conflicting* is not changed because the predicate in the schema

Conflicting_Set is not met for any new member of the set *Precedes*. Therefore, the set *Conflicting_Points* is also unchanged.

2 The operation conflicts with one or more operations in the history h that are in different transactions.

(i) Extra edges are added to the serialization graph that link the nodes representing the transactions containing the conflicting operations to the node representing transaction k if edges are not already in place.

If any edge completes a cycle, then a directed path links k with itself, thereby indicating that the history

$$h \frown opk$$

is non serializable.

(ii) The set *Precedes* has new members as in case (1).

The set *Conflicting* will have new members of the form (j, k), where j is the number of a transaction that contains an operation that conflicts with opk.

New elements, (i, k), are added to the set *Conflicting_Points* if there are now members

$$(i, j) \in Conflicting\_Points \wedge (j, k) \in Conflicting\_Points$$

The extra members correspond to the nodes being connected by the extra edges added to the serialization graphs in (i) above.

Should there already exist a member such that

$$(k, j) \in Conflicting\_Points$$

indicating a partial loop, then adding (j, k) to the set *Conflicting_Points* means that

$$(k, k) \in Conflicting\_Points$$

Thereby violating the invariant of the schema Conflicting_Trans and

indicating that

$h \frown opk$

is non serializable under the same conditions as in the serialization graph.

The above proof sketch demonstrates the direct correspondence between serialization graphs and the set *Conflicting_Points* for the same history record.

5.1.3 One Copy Serialization Property

To represent the property of one copy serialization, extra edges are added to the serialization graph to form the replicated data serialization graph, see Section 4.2.3.

The initial step in defining the one copy serialization property is represented by the schema Conflicting_Op_Set that defines the set of all transactions that conflict.

The set *Conflicting_Op* below contains all the paths in the form of pairs of transaction numbers representing the end points of the paths. The set is constructed by adding members to the set *Conflicting_Op*. Additional members are included for the following two reasons:

1 The transactions have operations that write to the same logical object.

2 There are three transactions, *T1*, *T2* and *T3*, such that *T1* reads a value of a logical data object after *T2* has written to the same logical data object, an operation in *T3* writes to the same logical data object and, finally, there is a conflicting relation between transactions *T2* and *T3* such that *T2* must occur before *T3*. If these conditions are satisfied, there is an addition constraint that transaction *T1* must occur before transaction *T3* and is represented by the additional member *(T1, T3)*.

Conflicting_Op_Set

Conflicting_Trans

Conflicting_Op : ℙ (Trans_Num × Trans_Num)

Conflicting_Op =

Conflicting_Points ∪

{o1, o2 : Op; op1, op2 : Operator; num1, num2 : Trans_Num;

obj1, obj2 : Physical_Object; v1, v2 : VALUE |

o1 = access (((op1, num1), obj1), v1) ∧

o2 = access (((op2, num2), obj2), v2) ∧ op1 = write ∧

op2 = write ∧ first obj1 = first obj2 ∧ (o1, o2) ∈ Precedes •

(num1, num2)} ∪

{o1, o2, o3 : Op; op1, op2, op3 : Operator;

num1, num2, num3 : Trans_Num;

obj1, obj2, obj3 : Physical_Object; v1, v2, v3 : VALUE |

o1 = access (((op1, num1), obj1), v1) ∧

o2 = access (((op2, num2), obj2), v2) ∧

o3 = access (((op3, num3), obj3), v3) ∧ (o2, o1) ∈ Precedes ∧

op1 = read ∧ op2 = write ∧ first obj1 = first obj2 ∧

o3 ∈ ran History_Rec ∧ op3 = write ∧

first obj3 = first obj1 ∧ num3 ≠ num1 ∧ num3 ≠ num2 ∧

(num2, num3) ∈ Conflicting_Points • (num1, num3)}

The schema Ordered_End_Points below defines the set *Ordered_Points* to contain all the transactions that have operations that conflict in terms of logical data objects.

Ordered_End_Points

Conflicting_Op_Set

Ordered_Points : ℙ (Trans_Num × Trans_Num)

Ordered_Points =

Conflicting_Op ∪

{n1, n2, n3 : Trans_Num |

(n1, n3) ∈ Ordered_Points ∧ (n3, n2) ∈ Ordered_Points •

(n1, n2)}

Finally, the one copy serialization property is expressed in the following schema.

One_Copy_Serialization

Ordered_End_Points

Invariant for the one copy seriability property

∀ n : Trans_Num • (n, n) ∉ Ordered_Points

Proof Sketch for the Representation of Replicated Data Serialization Graphs

The proof sketch that demonstrates that the schema One_Copy_Serialization correctly represents a replicated data serialization graph is similar to that used for serialization graphs.

The differences between the serialization graphs and replicated data serialization graphs are embodied in the schema Conflicting_Op_Set. This schema forms the set *Conflicting_Op* from the set *Conflicting_Points* and additional members. The additional members correspond to the extra edges being added to serialization graphs to

form replicated data serialization graphs, see Section 4.2. Thereby establishing a direct correspondence between the replicated data serialization graphs and the set *Ordered_Points* similar to the direct correspondence between the serialization graph and the set *Conflicting_Points*.

The replicated data serialization graphs must refer to complete histories. The following schema Complete_History restricts all histories to contain either a commit or an abort as the last operator for each transaction.

Complete_History

History_Rec : seq Op

$\forall$ o1 : Op; op1 : Operator; num1 : Trans_Num;
obj1 : Physical_Object; v1 : VALUE |
o1 = access (((op1, num1), obj1), v1) •
o1 $\in$ ran History_Rec $\Rightarrow$
($\exists$ o2 : Op; op2 : Operator; num2 : Trans_Num |
o2 = end (op2, num2) •
num1 = num2 $\wedge$
History_Rec^{-1} o1 < History_Rec^{-1} o2)

The one copy serialization property for complete histories is specified as the schema Complete_Serializable below.

Complete_Serializable $\triangleq$ One_Copy_Serialization $\wedge$ Complete_History

5.1.4 One Copy Serialization Histories

This subsection defines a schema that represents the set of all one copy serializable histories of operations based on the partial ordering provided by the set *Trans_Precedence* provided for the schema History_Invariant.

Serializable_History

Complete_Serializable

Histories : ℙ seq Op

Histories =

{h : seq Op | Complete_Serializable$_{[h/History_Rec]}$ • h}

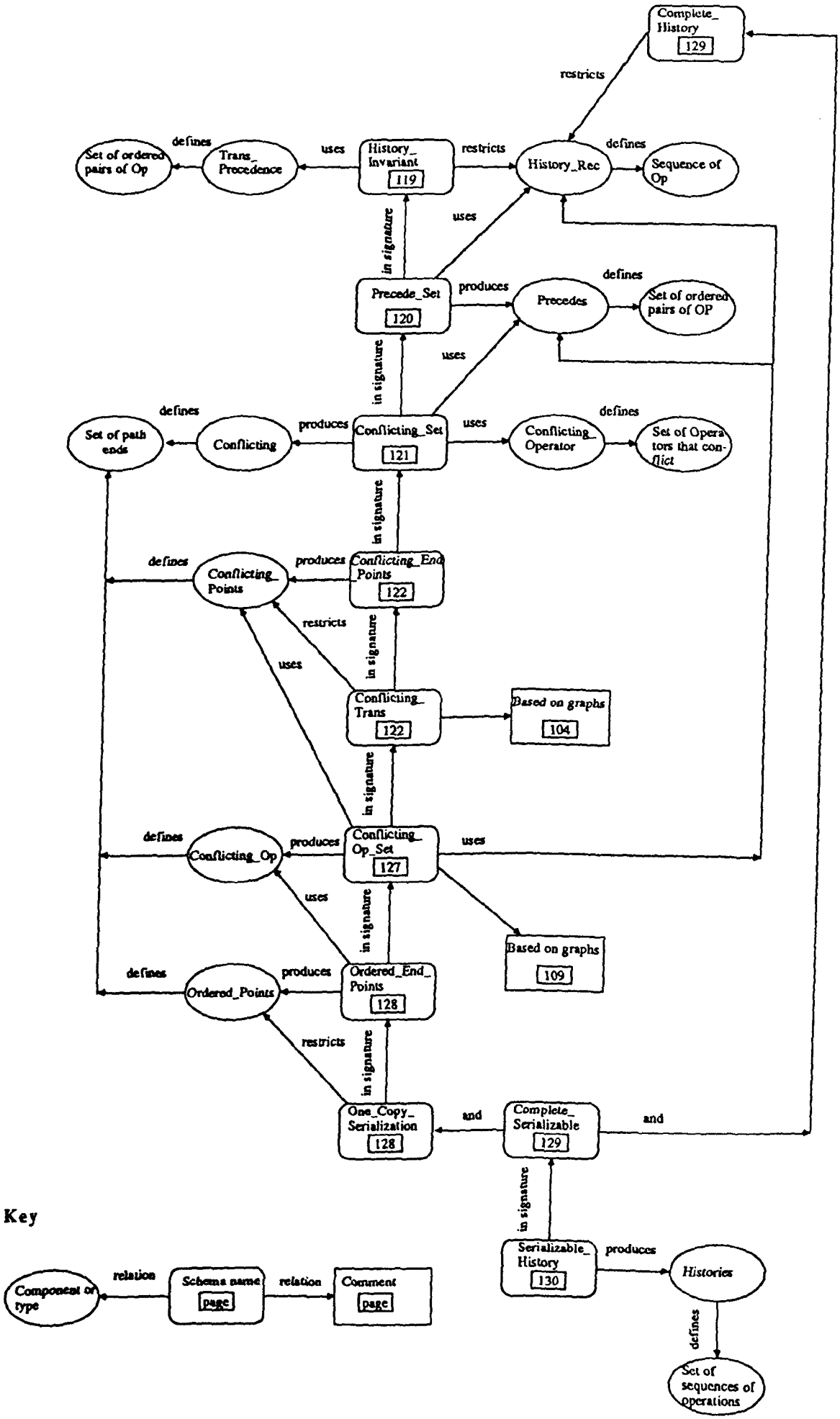
The schema Serializable_History does not include details of how the set *Histories* is constructed.

The notation used for renaming is described in reference [Wood89B] and takes the form in CADiZ of $Schema\_Id_{[new\_id\ /\ old\_id]}$.

Any history that is a member of the set *History* conforms to the predicate part of the schema Complete_Serializable, but with the variable *History_Rec* renamed by the bound variable *h*. The declarations in the schema Serializable_History providing the declarations of variables in the predicate part of the schema Complete_Serializable.

The final schema Serializable_History is constructed from several intermediate schemas and Figure 5.1 illustrates the inclusion relationship between the schemas used in the final expression of the one copy serialization property. The symbols used in the structure diagram in Figure 5.1 are similar to those used in the interaction diagrams in Chapter 3. However, in Figure 5.1 it is convenient to label the arrows to indicate the relation between two schemas or between schemas and components, instead of indicating the relation by comments in boxes.

Figure 5.1 Construction of the One Copy Serialization Property Schema



5.1.5 Initial State

A feasible initial state is specified by the schema Initial_Serializability_State below.

Initial_Serializability_State

Serializable_History'

History_Rec' = ⟨⟩

Proof Obligation for the Initial State

An empty sequence is a valid binding for *History_Rec*, hence an initial state does exist.

5.2 Implementation of a Replicated Database System

The implementation described in this section incorporates a basic two phase locking method at each site to ensure that no conflicts occur, see Section 4.3.1. The scheduling mechanism represented in the Z notation in this section is a strict two phase locking scheme in which the locks are not released until the transaction has completed all its operations.

If there is an attempt to execute a conflicting operation on an object that is waiting for a transaction to commit, the other operation is delayed until the conflict is resolved by the first transaction committing its operations. Should a deadlock condition arise then the scheduler has to abort one or more transaction and the transactions are repeated. The possibility of deadlocks is not addressed by the schemas.

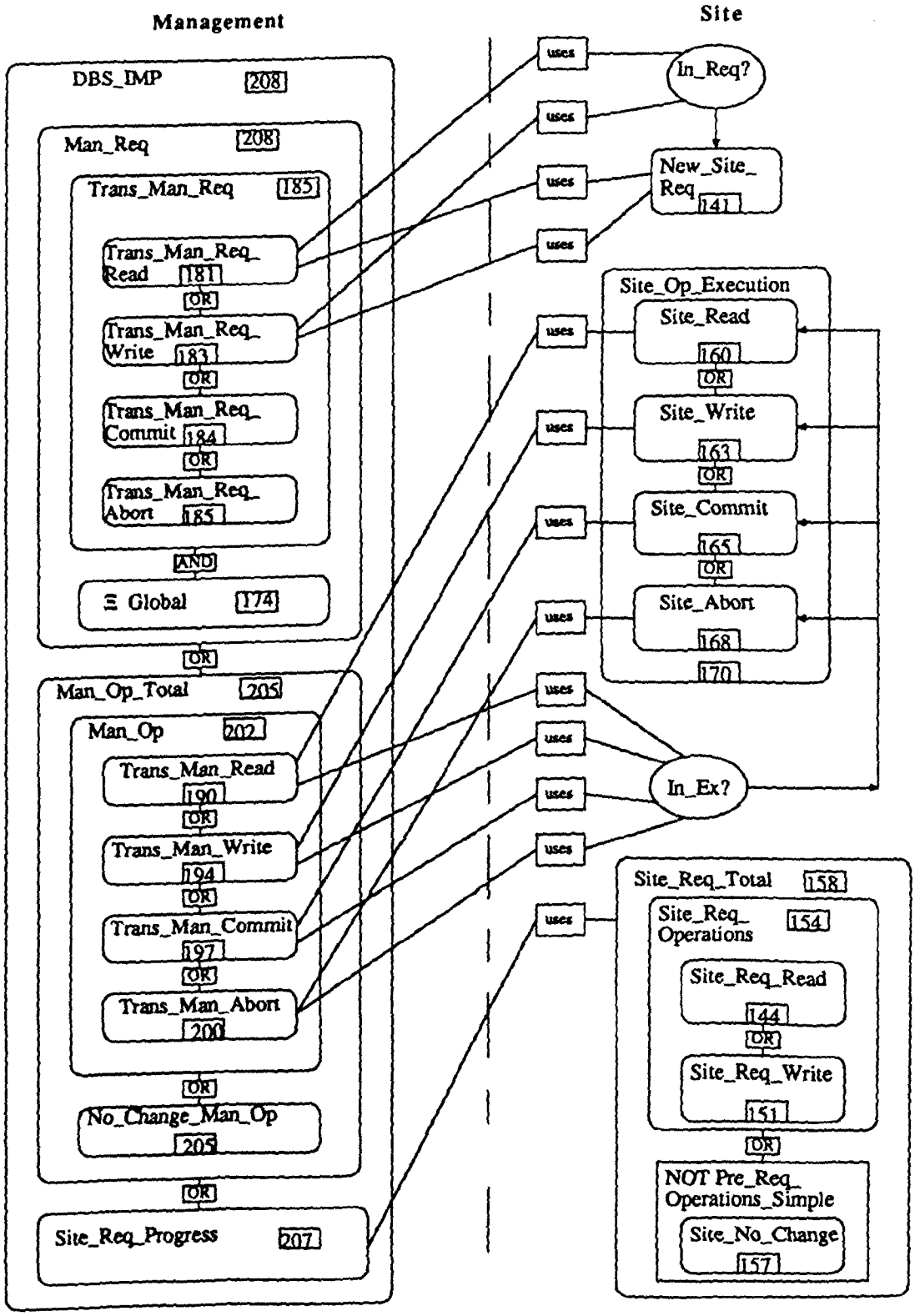
The schemas that model an implementation of a replicated database system are described in the following three sections. Section 5.3 deals with the site operations in four subsections. Similarly, Section 5.4 deals with the transaction management operations in four subsections. Finally, Section 5.5 contains a statement of the complete implementation of the

system. These sections contain all the component schemas in the implementation of the concurrency control aspects of a replicated database system. The definitions of the schemas and preconditions are included in the subsections to give an understanding of the complete description in the Z notation. However, the main features of the implementation can be appreciated without studying the schemas in detail.

The model of the behaviour of a replicated database system uses the concept of logical operations being received by a central management component of the replicated database system. The central management component translates the logical operations into one or more physical operations which are communicated to the distributed site components of the model. The sites then communicate their ability to perform the operations back to the central management component. The management component forms the physical operations that construct the historical record and form input variables for the site schemas.

The overall flow of control of the schemas is that a logical operation is treated as an input by the management request schemas. The management schemas change the state of the site request schemas by binding a new value to an input variable. The site request schemas respond to these inputs by updating their state variables. The changes to state variables cause the site request schemas to update the state variables of the management schemas. The new state values result in the management execution schemas constructing physical operation values that are incorporated both in a history representing the execution of physical operations and in the values of the input variables of the site execution schemas. The site execution schemas respond to these changes, to model the effects of the physical operations. Figures 5.2 illustrates the interaction between the management and site sets of schemas.

Figure 5.2 Interaction between the Management and Site Schemas



5.3 Site Operations

The schemas defined in this section fall into four categories:

1 The invariants for site schemas, Section 5.3.1.

2 The site schema for receiving requests for new physical operations, Section 5.3.2.

3 The schemas for defining the response to requests for access operations, Section 5.3.3.

4 The schemas for defining the response to execution of the operations, Section 5.3.4.

Before defining the schemas that model the site operations it is necessary to define an axiomatic schema that specifies an operator required by subsequent schemas.

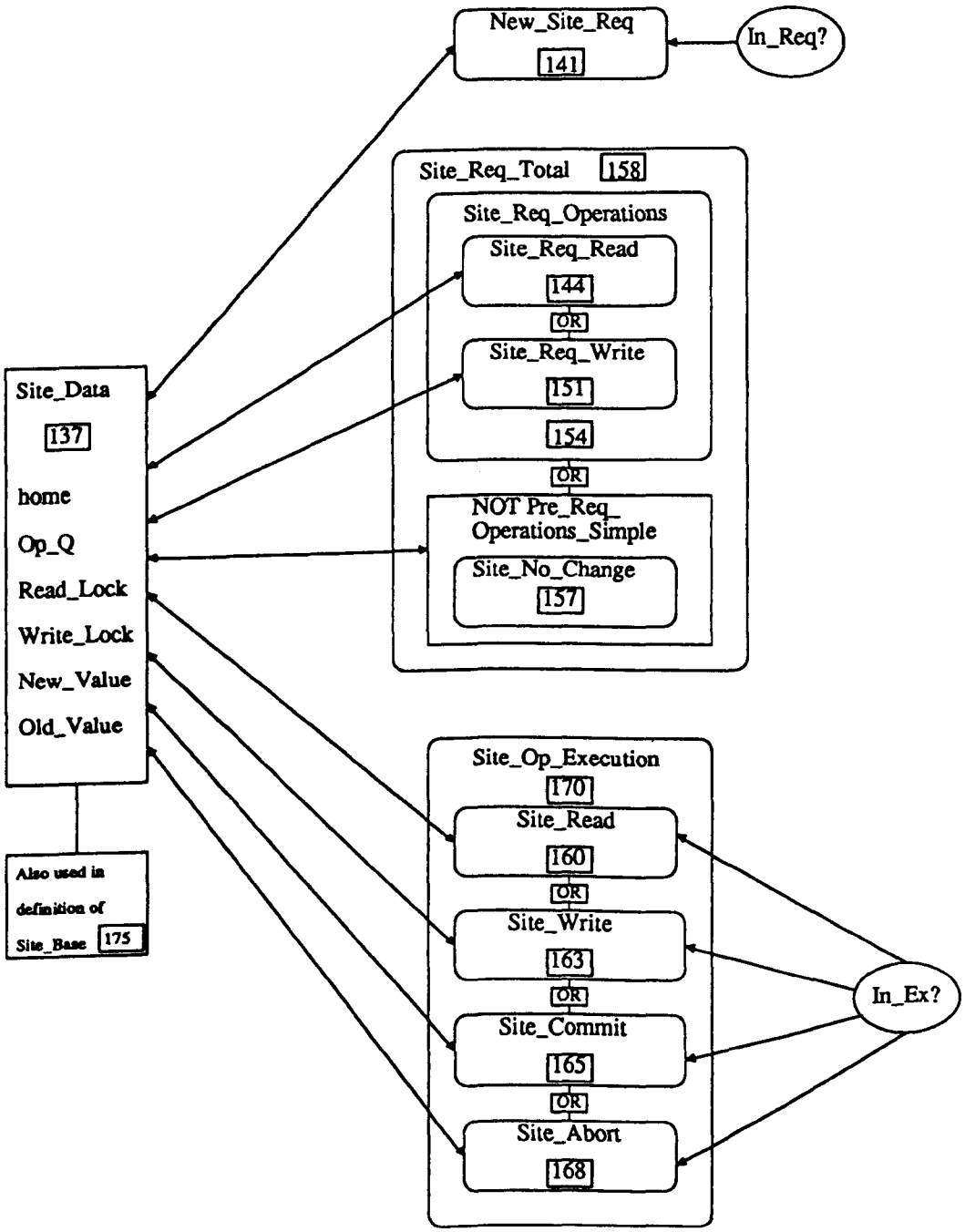
$$
\begin{array}{|l}
[X] \\
\hline
squash : (\mathbb{N} \pfun X) \fun seq\ X \\
\hline
squash\ \{\} = \langle\rangle \\
\forall f : \mathbb{N} \pfun X; i : \mathbb{N} \mid f \neq \{\} \land i = min\ (dom\ f) \bullet \\
\quad squash\ f = \langle f\ i \rangle \frown squash\ (\{i\} \ndres f) \\
\\
\end{array}
$$

The operator squash converts a function into a sequence of elements [Wood88] and is used to recreate a sequence of elements after some have been removed.

5.3.1 Invariants for Site Schemas

Figure 5.3 shows the basic interactions between the operation site schemas and the schema Site_Data that defines the state of each site.

Figure 5.3 Interactions between Site State Schema and Site Operation Schemas



The basic invariants of the site schemas are defined by the schema Site_Data below.

Site_Data

$$
\begin{aligned}
&home : Site \\
&Op\_Q : \mathrm{seq}\ Op \\
&Read\_Lock : \mathbb{P}\ Op \\
&Write\_Lock : \mathbb{P}\ Op \\
&New\_Value : Physical\_Object \rightarrow VALUE \\
&Old\_Value : Physical\_Object \rightarrow VALUE
\end{aligned}
$$

The only objects that have both read and write locks are in the same transaction

$$
\begin{aligned}
&\forall\ o1, o2 : Op;\ op1, op2 : Operator;\ num1, num2 : Trans\_Num; \\
&\quad obj1, obj2 : Physical\_Object;\ v1, v2 : VALUE \mid \\
&\quad o1 = access\ (((op1, num1), obj1), v1) \wedge \\
&\quad o2 = access\ (((op2, num2), obj2), v2) \wedge o1 \neq o2 \wedge \\
&\quad o1 \in Read\_Lock \wedge o2 \in Write\_Lock \wedge obj1 = obj2 \bullet \\
&\qquad num1 = num2
\end{aligned}
$$

If two operations have permission to write, then it is either to different physical objects or within the same transaction

$$
\begin{aligned}
&\forall\ o1, o2 : Op;\ op1, op2 : Operator;\ num1, num2 : Trans\_Num; \\
&\quad obj1, obj2 : Physical\_Object;\ v1, v2 : VALUE \mid \\
&\quad o1 = access\ (((op1, num1), obj1), v1) \wedge \\
&\quad o2 = access\ (((op2, num2), obj2), v2) \wedge o1 \neq o2 \wedge \\
&\quad o1 \in Write\_Lock \wedge o2 \in Write\_Lock \bullet \\
&\qquad obj1 \neq obj2 \vee num1 = num2
\end{aligned}
$$

The data type *Op_Q* records all operations referring to physical objects held at that site in which the operations have not been executed. A sequence ensures that the order in which

the operations are to be performed is conserved.

The data type *Read_Lock* maintains a record of all read operations that have been granted permission to read physical objects held at that site.

The data type *Write_Lock* maintains a record of the requests that have been granted to perform write operations.

The value of the variable *home* represents the identity of the site in which the instances of the schema Site_Data are currently bound. This allows different sites to be modelled by the same schema definition but with different values of the variable *home*. The reason for this component will become more apparent when the management schemas are defined. These later schemas treat the composite state of the system as containing a collection of independent subsets of values that correspond to individual sites. These subsets of state space can be interpreted loosely as objects in an object oriented approach to specification [Hall90], but they are 'weak' objects because they violate one important criterion of objects, in that their state is visible in the form of shared variables to other objects. However, familiarity with an object oriented approach to specification will allow the schema Site_Data to be viewed as an object, and the site request and site execution schemas representing operations performed on the objects. Using the concept of objects will help following the description of the implementation presented in this chapter.

The functions *Old_Value* and *New_Value* map physical data objects to their previous and current values respectively.

The first invariant specified by the schema Site_Data is that permits to read and write to the same data object cannot be given to different transactions. The second invariant is that permission cannot be given for two different operations to write to the same physical object unless they are within the same transaction.

The initial value held by a data object is defined below to have the type VALUE.

Initial : VALUE

An initial state, defined in the following schema, uses the global variable Initial as the default value held by data objects.

Initial_Site_State

Site_Data'

Op_Q' = ⟨⟩

Read_Lock' = ∅

Write_Lock' = ∅

Old_Value' = {d : Physical_Object • d ↦ Initial}

New_Value' = {d : Physical_Object • d ↦ Initial}

There is no information about the home variable

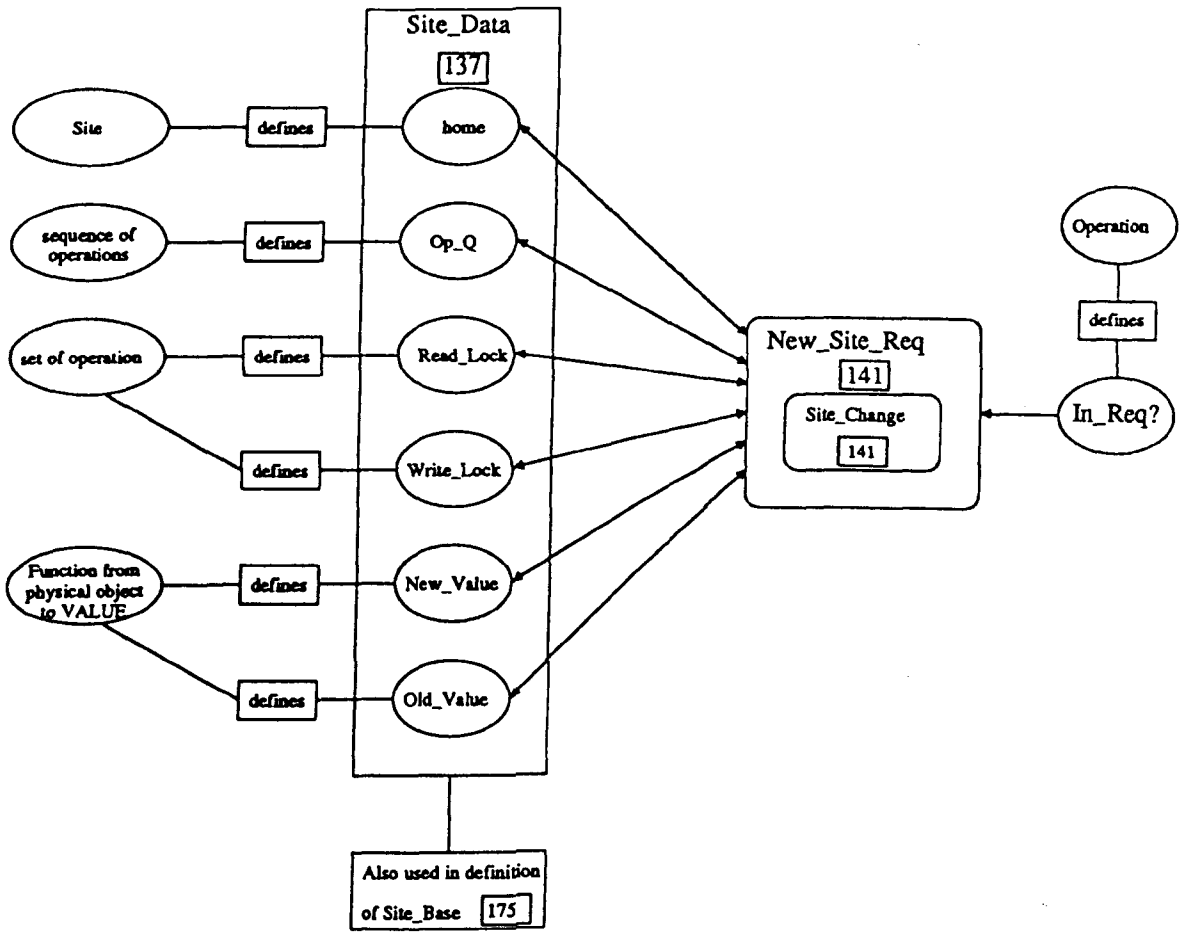
Proof Obligation for the Initial State

Each of the bindings for the components in the schema Site_Data are valid from their data definitions. In addition, none of the invariants in the predicate part of the schema Site_Data are violated.

5.3.2 Requests for Physical Operations

A diagrammatic view of the interactions involving the schemas New_Site_Req and Site_Data is given in Figure 5.4.

Figure 5.4 Schema for Site Requests



In all the site operations the variable *home* does not change value. This is represented by the schema Site_Change, which is included in all the site operation schemas.

Site_Change

ΔSite_Data

---

home' = home

The schema New_Site_Req is the only schema in the second category of site schema and it represents a receiver of physical operations as inputs from the management schemas.

New_Site_Req

Site_Change

In_Req? : Op

Op_Q' = Op_Q ⌢ ⟨In_Req?⟩

No other changes to the state variables

Read_Lock' = Read_Lock

Write_Lock' = Write_Lock

Old_Value' = Old_Value

New_Value' = New_Value

The schema New_Site_Req receives the physical operation as an input and adds it to the sequence of operations waiting to be handled by the site request schemas.

Proof Obligations for the Invariants of Schema New_Site_Req

The only component changed by the schema New_Site_Req is the sequence *Op_Q*. The type rules are obeyed and have been checked by CADiZ. The predicate of the

schema Site_Data does not use the sequence *Op_Q*, hence is not affected by the change.

Thus, the invariants are not violated by the schema New_Site_Req.

Preconditions of the Schema New_Site_Req

The only precondition for the schema New_Site_Req is that the components are of the correct type, in particular, *In_Req?* has the type Op, i.e. a physical operation. This is given by the schema Pre_New_Site_Req.

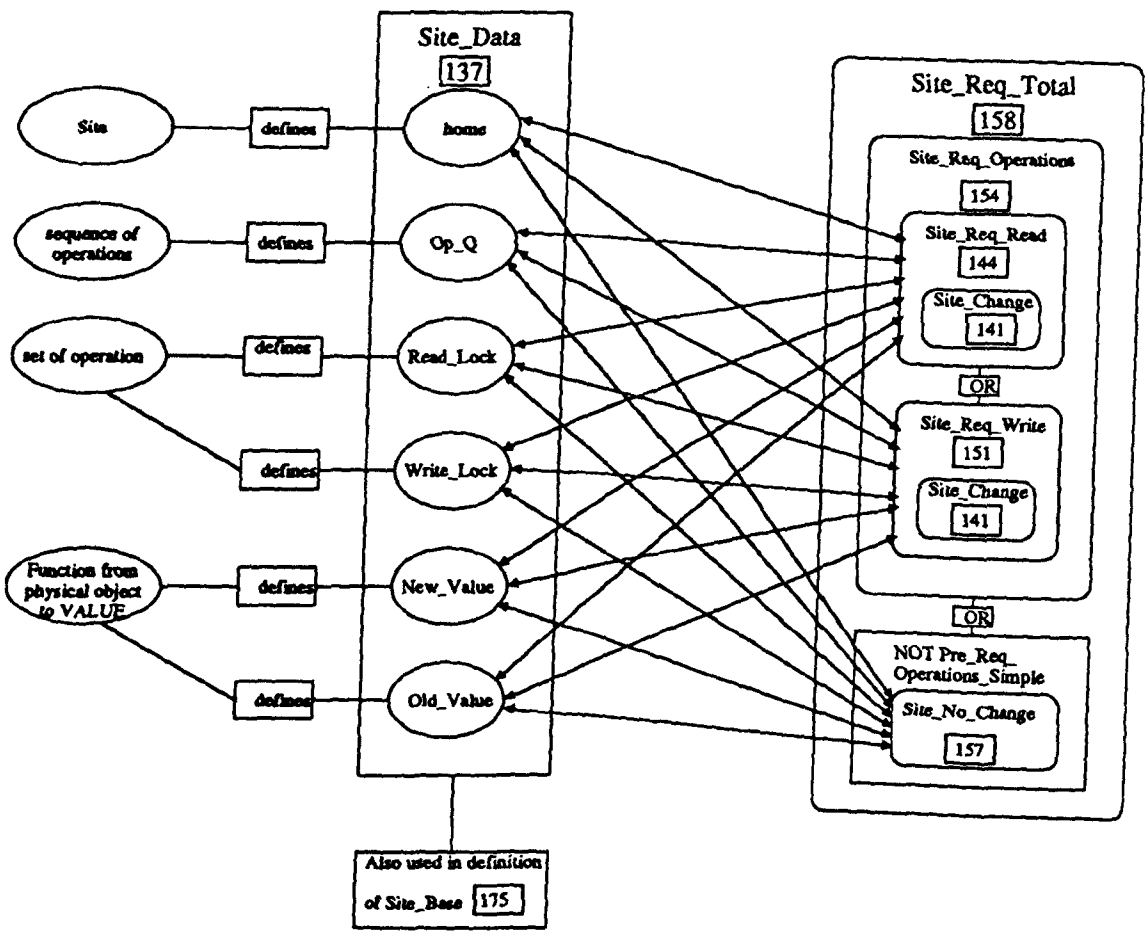
Pre_New_Site_Req

home : Site
Op_Q : seq Op
Read_Lock : ℙ Op
Write_Lock : ℙ Op
New_Value : Physical_Object ⟶ VALUE
Old_Value : Physical_Object ⟶ VALUE
In_Req? : Op

5.3.3 Site Response to Requests to Perform Operations

Figure 5.5 shows the interactions between the site request schemas and the site data schema.

Figure 5.5 Schemas for Site Response



The schema Site_Req_Read is one of the third category and represents the changes of state that occur as the result of state values only; no inputs are involved.

Site_Req_Read

Site_Change

$\exists$ o1 : Op; op1 : Operator; obj1 : Physical_Object;
v1 : VALUE; num1 : Trans_Num |
o1 = access (((op1, num1), obj1), v1) $\wedge$
home = second obj1 $\wedge$ op1 = read •

The operation is the first operation waiting to be executed for that transaction

o1 $\in$ ran Op_Q $\wedge$
($\forall$ o2 : Op; op2 : Operator; obj2 : Physical_Object;
num2 : Trans_Num; v2 : VALUE |
o2 = access (((op2, num2), obj2), v2) $\wedge$
o2 $\in$ ran Op_Q $\wedge$ num1 = num2 •
Op_Q^{-1} o1 < Op_Q^{-1} o2)

No other transaction has a conflicting lock on the same object

$\wedge$
$\neg$
($\exists$ o2 : Op; op2 : Operator; obj2 : Physical_Object;
num2 : Trans_Num; v2 : VALUE •
o2 = access (((op2, num2), obj2), v2) $\wedge$
num1 $\neq$ num2 $\wedge$ o2 $\in$ Write_Lock $\wedge$ obj2 = obj1) $\wedge$
Read_Lock' = Read_Lock $\cup$ {o1}

No change in other Site_Data declarations

Op_Q' = Op_Q
Write_Lock' = Write_Lock
New_Value' = New_Value

Old_Value' = Old_Value

The input operation is taken from the sequence *Op_Q* such that it is the first operation to be performed for that transaction and, if there is no write lock on the same physical object, the state is updated as indicated. That is, the operation is added to the set *Read_Lock* which is accessed by the management schemas.

Proof Obligation for the schema Site_Req_Read

The proof obligation is split into two parts:

(i) the type properties of the components are not violated

(ii) the predicate of schema Site_Data is not invalidated.

(i) The only component changed by the schema Site_Req_Read is the set *Read_Lock*. These changes conform to the type rules of the Z notation as interpreted by CADiℤ.

(ii) The set *Read_Lock* is changed so that

$$Read\_Lock' = Read\_Lock \cup \{o\,1\}$$

such that

$$\neg \exists\, o\,2 : Op \bullet o\,2 \in Write\_Lock \wedge object(o\,2) = object(o\,1) \wedge$$

$$number(o\,2) \neq number(o\,1)$$

where *object* is a projection function that extracts the physical object value from physical operations, hence has the type

$$Op \longrightarrow Physical\_Object$$

and *number* is a projection function that extracts the transaction number from physical operations, hence has the type

$Op \longrightarrow Trans\_Num$

Therefore, with

$o1 \in Read\_Lock'$

for all $o2$ such that

$o2 \in Write\_Lock \wedge object(o2) = object(o1)$

then from the predicate in the schema Site_Req_Read, the physical operations must have the same transaction numbers, i.e.

$number\ (o2) = number\ (o1)$

hence conforming to the first universally quantified predicate in the schema Site_Data.

The set *Write_Lock* is not changed, hence cannot invalidate the second universally quantified predicate in the schema Site_Data.

Note that the operation selected from the range of Op_Q is not unique. The state of Op_Q could be such that several members in its range meet the preconditions, in such cases the value chosen is non deterministic. When viewed in isolation from other schemas, this non determinism has the advantage of not over specifying the requirements. However, when schema terms are connected together, there can be cases when multiple preconditions are true simultaneously, thereby causing ambiguity about the changes in state. To overcome any possible difficulties an extra component can be included in the schema declaration part that externally identifies the operation that takes place. The advantages of including an extra component in the schemas for this purpose are not important in this study, therefore no such extra components are used.

Preconditions for the Schema Site_Req_Read

Preconditions for the expanded schema Site_Req_Read are expressed by the following schema.

Pre_Req_Read_Expand

home : Site

Op_Q : seq Op

Read_Lock : $\mathbb{P}$ Op

Write_Lock : $\mathbb{P}$ Op

New_Value : Physical_Object $\rightarrow$ VALUE

Old_Value : Physical_Object $\rightarrow$ VALUE

$\exists$ home' : Site; Op_Q' : seq Op; Read_Lock' : $\mathbb{P}$ Op;
Write_Lock' : $\mathbb{P}$ Op;
New_Value' : Physical_Object $\rightarrow$ VALUE;
Old_Value' : Physical_Object $\rightarrow$ VALUE •
($\exists$ o1 : Op; op1 : Operator; obj1 : Physical_Object;
v1 : VALUE; num1 : Trans_Num |
o1 = access (((op1, num1), obj1), v1) $\wedge$
home = second obj1 $\wedge$ op1 = read •
o1 $\in$ ran Op_Q $\wedge$
($\forall$ o2 : Op; op2 : Operator; obj2 : Physical_Object;
num2 : Trans_Num; v2 : VALUE |
o2 = access (((op2, num2), obj2), v2) $\wedge$
o2 $\in$ ran Op_Q $\wedge$ num1 = num2 •
Op_Q^{-1} o1 < Op_Q^{-1} o2) $\wedge$
$\neg$
($\exists$ o2 : Op; op2 : Operator; obj2 : Physical_Object;
num2 : Trans_Num; v2 : VALUE •
o2 = access (((op2, num2), obj2), v2) $\wedge$
num1 $\neq$ num2 $\wedge$ o2 $\in$ Write_Lock $\wedge$

obj2 = obj1) ∧

Read_Lock' = Read_Lock ∪ {o1}) ∧

Op_Q' = Op_Q ∧ Write_Lock' = Write_Lock ∧

New_Value' = New_Value ∧ Old_Value' = Old_Value ∧

home' = home ∧

(∀ o1, o2 : Op; op1, op2 : Operator;

num1, num2 : Trans_Num; obj1, obj2 : Physical_Object;

v1, v2 : VALUE |

o1 = access (((op1, num1), obj1), v1) ∧

o2 = access (((op2, num2), obj2), v2) ∧ o1 ≠ o2 ∧

o1 ∈ Read_Lock ∧ o2 ∈ Write_Lock ∧ obj1 = obj2 •

num1 = num2) ∧

(∀ o1, o2 : Op; op1, op2 : Operator;

num1, num2 : Trans_Num; obj1, obj2 : Physical_Object;

v1, v2 : VALUE |

o1 = access (((op1, num1), obj1), v1) ∧

o2 = access (((op2, num2), obj2), v2) ∧ o1 ≠ o2 ∧

o1 ∈ Write_Lock ∧ o2 ∈ Write_Lock •

obj1 ≠ obj2 ∨ num1 = num2) ∧

(∀ o1, o2 : Op; op1, op2 : Operator;

num1, num2 : Trans_Num; obj1, obj2 : Physical_Object;

v1, v2 : VALUE |

o1 = access (((op1, num1), obj1), v1) ∧

o2 = access (((op2, num2), obj2), v2) ∧ o1 ≠ o2 ∧

o1 ∈ Read_Lock' ∧ o2 ∈ Write_Lock' ∧ obj1 = obj2 •

num1 = num2) ∧

(∀ o1, o2 : Op; op1, op2 : Operator;

num1, num2 : Trans_Num; obj1, obj2 : Physical_Object;

v1, v2 : VALUE |

o1 = access (((op1, num1), obj1), v1) ∧

o2 = access (((op2, num2), obj2), v2) ∧ o1 ≠ o2 ∧

o1 ∈ Write_Lock' ∧ o2 ∈ Write_Lock' •

obj1 ≠ obj2 ∨ num1 = num2)

The preconditions are simplified to give the schema Pre_Req_Read_Simple below.

Pre_Req_Read_Simple

Site_Data

$$
\begin{aligned}
&\exists\, o1 : Op;\ obj1 : Physical\_Object;\ v1 : VALUE;\\
&\quad num1 : Trans\_Num \mid\\
&\quad o1 = access\ (((read, num1), obj1), v1)\ \wedge\\
&\quad home = second\ obj1\ \bullet\\
&\qquad o1 \in ran\ Op\_Q\ \wedge\\
&\qquad (\forall\, o2 : Op;\ op2 : Operator;\ obj2 : Physical\_Object;\\
&\qquad\quad num2 : Trans\_Num;\ v2 : VALUE \mid\\
&\qquad\quad o2 = access\ (((op2, num2), obj2), v2)\ \wedge\\
&\qquad\quad o2 \in ran\ Op\_Q\ \wedge\ num1 = num2\ \bullet\\
&\qquad\qquad Op\_Q^{-1}\ o1 < Op\_Q^{-1}\ o2)\ \wedge\\
&\qquad \neg\\
&\qquad (\exists\, o2 : Op;\ op2 : Operator;\ num2 : Trans\_Num;\\
&\qquad\quad v2 : VALUE\ \bullet\\
&\qquad\qquad o2 = access\ (((op2, num2), obj1), v2)\ \wedge\\
&\qquad\qquad num1 \neq num2\ \wedge\ o2 \in Write\_Lock)
\end{aligned}
$$

This leads to the following schema which expresses the equivalence between the simplified preconditions and the rudimentary preconditions given by the *pre* operator.

$$Simplified\_5\_1 \triangleq pre\ Site\_Req\_Read \Leftrightarrow Pre\_Req\_Read\_Simple$$

Note that an input operation is not required for a change of state.

The change of state as a result of analysing a write request is described by the following schema Site_Req_Write.

Site_Req_Write

Site_Change

$\exists$ o1 : Op; op1 : Operator; obj1 : Physical_Object;
v1 : VALUE; num1 : Trans_Num |
o1 = access (((op1, num1), obj1), v1) $\wedge$
home = second obj1 $\wedge$ op1 = write •

The operation is the first one waiting to be executed for that transaction

o1 $\in$ ran Op_Q $\wedge$
($\forall$ o2 : Op; op2 : Operator; obj2 : Physical_Object;
num2 : Trans_Num; v2 : VALUE |
o2 = access (((op2, num2), obj2), v2) $\wedge$
o2 $\in$ ran Op_Q $\wedge$ num1 = num2 •
Op_Q^{-1} o1 < Op_Q^{-1} o2)

There are no conflicting locks on the same data object

$\wedge$
$\neg$
($\exists$ o2 : Op; op2 : Operator; obj2 : Physical_Object;
num2 : Trans_Num; v2 : VALUE •
o2 = access (((op2, num2), obj2), v2) $\wedge$
num1 $\neq$ num2 $\wedge$ o2 $\in$ Read_Lock $\cup$ Write_Lock $\wedge$
obj2 = obj1) $\wedge$ Write_Lock' = Write_Lock $\cup$ {o1}

No change in value for other declarations in Site_Data

Op_Q' = Op_Q
Read_Lock' = Read_Lock
New_Value' = New_Value

Old_Value' = Old_Value

The Site_Req_Write schema obtains an operation which is a member of the range of the sequence *Op_Q* and, if there are no locks to read or write to the same physical object, changes the state of the site variables.

Proof Obligation for the Schema Site_Req_Write

Similar to the proof sketch for the schema Site_Req_Read, the proof obligation is split into two parts:

(i) the type properties of the components are not violated

(ii) the predicate of schema Site_Data is not invalidated.

(i) The only component changed by the schema Site_Req_Write is the set *Write_Lock*. These changes conform to the type rules of the Z notation as interpreted by CADiℤ.

(ii) The set *Write_Lock* is changed so that

$$Write\_Lock' = Write\_Lock \cup \{o\,1\}$$

such that

$$\neg \exists\, o\,2 : Op \bullet o\,2 \in Read\_Lock \cup Write\_Lock \wedge$$

$$object(o\,2) = object(o\,1) \wedge number(o\,2) \neq number(o\,1)$$

Therefore, with

$$o\,1 \in Write\_Lock'$$

for all *o2* such that

$$o\,2 \in Read\_Lock \wedge object(o\,2) = object(o\,1)$$

then the physical operations must have the same transaction numbers, i.e.

$number(o2) = number(o1)$

hence conforming to the first universally quantified predicate in the schema Site_Data.

Similarly, for

$o1 \in Write\_Lock'$

for all $o2$ such that

$o2 \in Write\_Lock$

then from the predicate in the schema Site_Req_Write the physical operations must either have the same transaction numbers or be to different physical objects, i.e.

$object(o2) \neq object(o1) \vee number(o2) = number(o1)$

hence conforming to the second universally quantified predicate in the schema Site_Data.

Preconditions for the Schema Site_Req_Write

The preconditions for the schema Site_Req_Write are simplified in the following schema.

Pre_Req_Write_Simple

Site_Data

$$
\begin{aligned}
&\exists\, o1 : Op;\ obj1 : Physical\_Object;\ v1 : VALUE;\\
&\quad num1 : Trans\_Num \mid\\
&\quad o1 = access(((write, num1), obj1), v1) \wedge\\
&\quad home = second\ obj1 \bullet\\
&\qquad o1 \in ran\ Op\_Q \wedge\\
&\qquad (\forall\, o2 : Op;\ op2 : Operator;\ obj2 : Physical\_Object;\\
&\qquad\quad num2 : Trans\_Num;\ v2 : VALUE \mid\\
&\qquad\quad o2 = access(((op2, num2), obj2), v2) \wedge\\
&\qquad\quad o2 \in ran\ Op\_Q \wedge num1 = num2 \bullet\\
&\qquad\qquad Op\_Q^{-1}\ o1 < Op\_Q^{-1}\ o2) \wedge\\
&\qquad \neg\\
&\qquad (\exists\, o2 : Op;\ op2 : Operator;\ num2 : Trans\_Num;\\
&\qquad\quad v2 : VALUE \bullet\\
&\qquad\qquad o2 = access(((op2, num2), obj1), v2) \wedge\\
&\qquad\qquad num1 \neq num2 \wedge o2 \in Read\_Lock \cup Write\_Lock)
\end{aligned}
$$

The correctness of the preconditions is expressed in terms of the following schema.

$$Simplified\_5\_2 \triangleq pre\ Site\_Req\_Write \Leftrightarrow Pre\_Req\_Write\_Simple$$

The two request site schemas are combined to form the schema Site_Req_Operations.

$$Site\_Req\_Operations \triangleq Site\_Req\_Read \vee Site\_Req\_Write$$

Because both the schemas define uniquely the postconditions of the operation, any change of state is the consequence of one of the schemas. The preconditions of the two site request

schemas are not mutually exclusive because different bindings of objects can be used for the existentially quantified variable *o1*. Using the rule:

$$\exists P(x) \vee \exists Q(x) \Leftrightarrow \exists (P(x) \vee Q(x))$$

The preconditions identify the conditions under which the disjunction of the two schemas is true. When the combined preconditions are simplified to use the same quantified variable *o1* it gives the impression that the preconditions of each operation cannot be simultaneously true. However, this is not the case because the bound variables in the operation schemas can take different values independently of each other.

The preconditions for the schema Site_Req_Operations are simplified to the schema Pre_Req_Operations_Simple below.

Pre_Req_Operations_Simple

Site_Data

$\exists$ o1 : Op; obj1 : Physical_Object; v1 : VALUE;
num1 : Trans_Num •
o1 = access (((read, num1), obj1), v1) $\wedge$
home = second obj1 $\wedge$ o1 $\in$ ran Op_Q $\wedge$
($\forall$ o2 : Op; op2 : Operator; obj2 : Physical_Object;
num2 : Trans_Num; v2 : VALUE |
o2 = access (((op2, num2), obj2), v2) $\wedge$
o2 $\in$ ran Op_Q $\wedge$ num1 = num2 •
Op_Q^{-1} o1 < Op_Q^{-1} o2) $\wedge$
$\neg$
($\exists$ o2 : Op; op2 : Operator; num2 : Trans_Num;
v2 : VALUE •
o2 = access (((op2, num2), obj1), v2) $\wedge$
num1 $\neq$ num2 $\wedge$ o2 $\in$ Write_Lock)

Read request

$\vee$

o1 = access (((write, num1), obj1), v1) $\wedge$
home = second obj1 $\wedge$ o1 $\in$ ran Op_Q $\wedge$
($\forall$ o2 : Op; op2 : Operator; obj2 : Physical_Object;
num2 : Trans_Num; v2 : VALUE |
o2 = access (((op2, num2), obj2), v2) $\wedge$
o2 $\in$ ran Op_Q $\wedge$ num1 = num2 •
Op_Q^{-1} o1 < Op_Q^{-1} o2) $\wedge$

$$\neg (\exists o2 : Op;\ op2 : Operator;\ num2 : Trans\_Num;\ v2 : VALUE \bullet o2 = access(((op2, num2), obj1), v2) \land num1 \neq num2 \land o2 \in Read\_Lock \cup Write\_Lock)$$

write request

The site request operations are not total since for some values of the state described by the schema Site_Data the preconditions are false. It is necessary to make explicit that there is no change of state when the site request operation occurs and the preconditions are false. First, a schema must be defined that represents no change of the site data.

Site_No_Change

ΔSite_Data

home' = home
Op_Q' = Op_Q
Read_Lock' = Read_Lock
Write_Lock' = Write_Lock
New_Value' = New_Value
Old_Value' = Old_Value

Proof Obligation for the Schema Site_No_Change

Since none of the components are not changed, the invariants must be maintained.

The notation ΞSite_Data cannot be used in this case because it causes problems with CADiZ later when the schema Site_Req_Total is expanded.

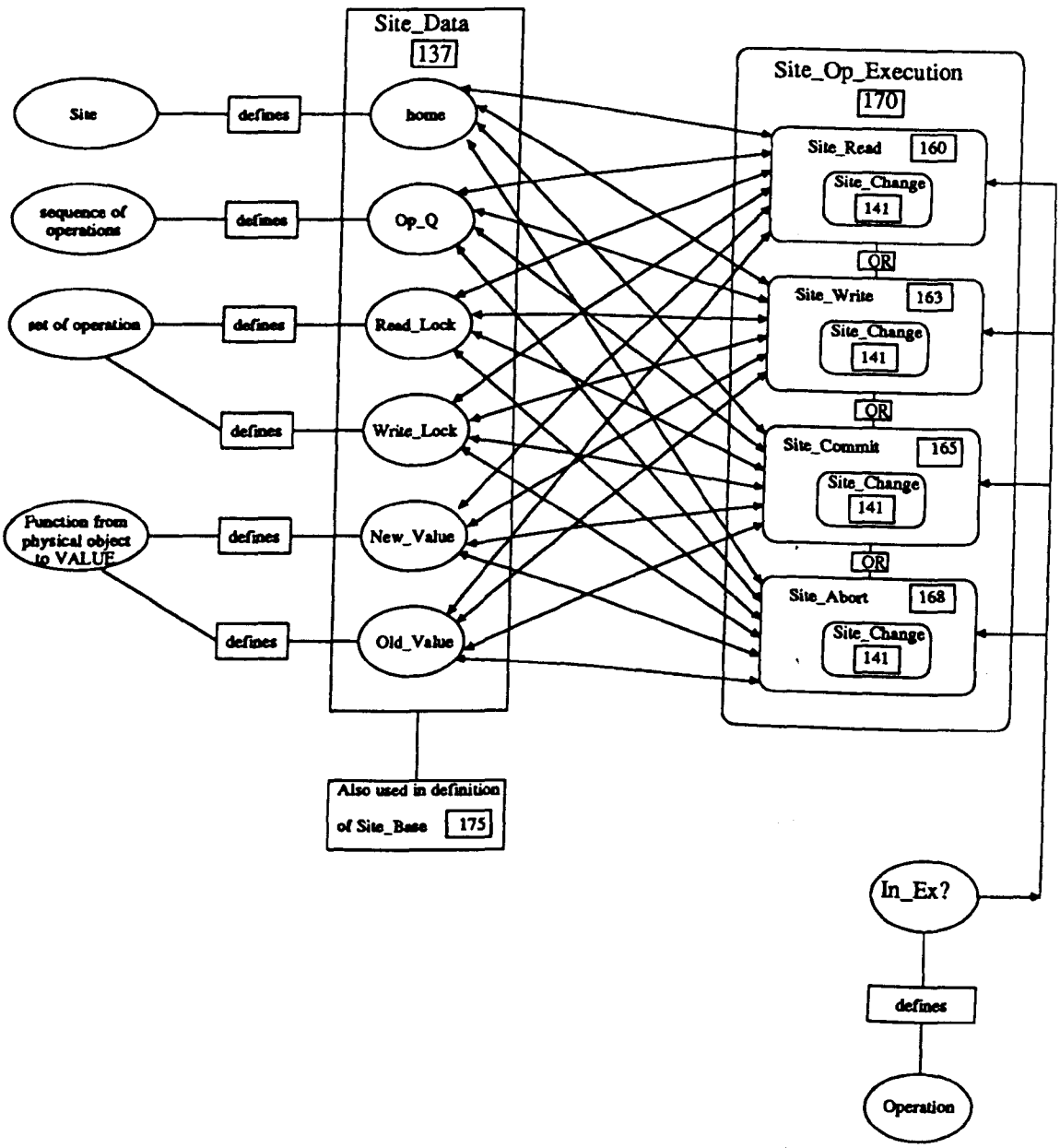
The total specification of the site request operation is:

$$\begin{aligned} &\text{Site\_Req\_Total} \mathrel{\widehat{=}} \\ &\quad \text{Site\_Req\_Operations} \lor \\ &\quad \neg\, \text{Pre\_Req\_Operations\_Simple} \land \text{Site\_No\_Change} \end{aligned}$$

5.3.4 Change of Site State in Response to Execution of Operations

The fourth category of site schemas are those that describe the execution of the operations that are triggered by the management schemas changing the values bound to the input variable for physical operations to be executed by a site. Figure 5.6 illustrates the interactions between the site execution schemas and the site data schema.

Figure 5.6 Schemas for Site Execution



The first schema of this category is for the read operation and is the schema Site_Read below.

Site_Read

Site_Change

In_Ex? : Op

$\exists$ p_site : Operator; b_site : Physical_Object;

n_site : Trans_Num; v_site : VALUE |

In_Ex? = access (((p_site, n_site), b_site), v_site) $\wedge$

home = second b_site $\wedge$ p_site = read •

v_site = New_Value b_site $\wedge$ In_Ex? $\in$ ran Op_Q $\wedge$

Op_Q' = squash (Op_Q $\rhd$ {In_Ex?})

No other changes to site data

$\wedge$ Read_Lock' = Read_Lock $\wedge$

Write_Lock' = Write_Lock $\wedge$ Old_Value' = Old_Value $\wedge$

New_Value' = New_Value $\vee$

(v_site $\neq$ New_Value b_site $\vee$ In_Ex? $\notin$ ran Op_Q)

No change to any site data variables if In_Ex? not in Op_Q

or the wrong value is given

$\wedge$ Op_Q' = Op_Q $\wedge$ Read_Lock' = Read_Lock $\wedge$

Write_Lock' = Write_Lock $\wedge$ Old_Value' = Old_Value $\wedge$

New_Value' = New_Value

The schema Site_Read uses the value of the input variable *In_Ex?* for initiating a change of state. Should *In_Ex?* meet the preconditions of this schema, the physical operation is removed from the variable *Op_Q* to indicate that the operation has been executed. A read operation does not involve any other change of state. The value of the data object read by this operation is that given by the function *New_Value*, but the model of the

implementation does not describe how this value is communicated to the environment.

At this level of abstraction, the value read from the database is presented as a precondition. This implies that the input variable must include this value before it can read it, which is contrary to the expected normal sequence of events. It is presented in this manner because the actual transfer of data does not affect the concurrency control of the model of a database system, hence the details of how the management functions obtain the value are hidden from the view of the operation of database systems. Should the value not equal that given by the function *New_Value* then there is no change of state. It follows that the schema is total over all values of the *v_site* component. A value is required for the *v_site* component to identify the operation in the *Op_Q* component. A different approach is to change the definition of the read operation such that it does not have a VALUE component.

Proof Obligation for the Schema Site_Read

The only component to be changed under some conditions is the sequence *Op_Q*. The changes to this component do not impact on the predicate in the schema Site_Data and the changes obey the type rules for the Z notation. Therefore, the invariants are not violated by the schema Site_Read.

Preconditions for the Schema Site_Read

The preconditions for the schema Site_Read are simplified to the following schema.

Pre_Site_Read_Simple

Site_Data

In_Ex? : Op

Existentially quantifying components of In_Ex?

$\exists$ b_site : Physical_Object; n_site : Trans_Num;

v_site : VALUE •

In_Ex? = access (((read, n_site), b_site), v_site) $\wedge$

home = second b_site

Leading to the schema below that represents the equivalence between the simplified preconditions and the rudimentary preconditions.

$$\text{Simplified\_5\_3} \mathrel{\widehat{=}} \text{pre Site\_Read} \Leftrightarrow \text{Pre\_Site\_Read\_Simple}$$

The execution of the write operation is defined by the schema Site_Write below.

Site_Write

Site_Change

In_Ex? : Op

Existentially quantifying components of In_Ex?

$\exists$ p_site : Operator; b_site : Physical_Object;

n_site : Trans_Num; v_site : VALUE |

In_Ex? = access (((p_site, n_site), b_site), v_site) $\wedge$

home = second b_site $\wedge$ p_site = write •

In_Ex? $\in$ ran Op_Q $\wedge$

Op_Q' = squash (Op_Q $\rhd$ {In_Ex?}) $\wedge$

New_Value' = New_Value $\oplus$ {b_site $\mapsto$ v_site}

No other changes to variables in Site_Data

$\wedge$ Read_Lock' = Read_Lock $\wedge$

Write_Lock' = Write_Lock $\wedge$ Old_Value' = Old_Value $\vee$

In_Ex? $\notin$ ran Op_Q

No changes to the variables in Site_Data if In_Ex? is not in Op_Q

$\wedge$ Op_Q' = Op_Q $\wedge$ New_Value' = New_Value $\wedge$

Read_Lock' = Read_Lock $\wedge$ Write_Lock' = Write_Lock $\wedge$

Old_Value' = Old_Value

The schema Site_Write existentially quantifies the components of the variable *In_Ex?* and updates the current value of the specified data object, provided the input operation is in the sequence *Op_Q*. The physical write operation is also removed from the sequence of operations held in the variable *Op_Q*. There is no change of variables in the schema Site_Data if the input value is not in the sequence *Op_Q*.

Proof Obligation for the Schema Site_Write

In addition to the possible changes to the sequence *Op_Q*, the function *New_Value*

may also be updated. Neither *Op_Q* or *New_Value* impact on the predicate in the schema Site_Data, hence cannot invalidate the conditions.

The function *New_Value* is changed by using the functional override operator such that the overriding function is a single maplet. Thereby, conforming to the type requirements to ensure that the resultant type is a total function.

Preconditions for the Schema Site_Write

The schema Pre_Site_Write_Simple gives the simplied preconditions for the schema Site_Write.

Pre_Site_Write_Simple

$$
\begin{array}{|l}
\hline
Site\_Data \\
In\_Ex? : Op \\
\hline
\exists\, b\_site : Physical\_Object;\ n\_site : Trans\_Num; \\
\quad v\_site : VALUE \bullet \\
\qquad In\_Ex? = access\ (((write, n\_site), b\_site), v\_site) \land \\
\qquad home = second\ b\_site \\
\hline
\end{array}
$$

The correctness of the simplified preconditions is stated in the schema below.

$$Simplified\_5\_4 \triangleq pre\ Site\_Write \Leftrightarrow Pre\_Site\_Write\_Simple$$

The last two operations to be executed by the site schemas are the end operations of commit and abort.

The commit operation is represented by the schema Site_Commit.

Site_Commit

Site_Change

In_Ex? : Op

Existentially quantifying components

$\exists$ p_site : Operator; n_site : Trans_Num |

In_Ex? = end (p_site, n_site) $\wedge$ p_site = commit •

Old_Value' =

Old_Value $\oplus$

{o1 : Op; op1 : Operator; num1 : Trans_Num;

obj1 : Physical_Object; v1 : VALUE |

o1 = access (((op1, num1), obj1), v1) $\wedge$

o1 $\in$ Write_Lock $\wedge$ num1 = n_site •

obj1 $\mapsto$ New_Value obj1} $\wedge$

Read_Lock' =

Read_Lock \

{o1 : Op; op1 : Operator; num1 : Trans_Num;

obj1 : Physical_Object; v1 : VALUE |

o1 = access (((op1, num1), obj1), v1) $\wedge$ num1 = n_site •

o1} $\wedge$

Write_Lock' =

Write_Lock \

{o1 : Op; op1 : Operator; num1 : Trans_Num;

obj1 : Physical_Object; v1 : VALUE |

o1 = access (((op1, num1), obj1), v1) $\wedge$ num1 = n_site •

o1}

No change in the other declarations in Site_Data

$$Op\_Q' = Op\_Q$$

$$New\_Value' = New\_Value$$

The schema Site_Commit uses the operation determined by *In_Ex?* to update the components *Old_Value*, *Read_Lock* and *Write_Lock* for the transaction that has been committed.

Proof Obligation for the Schema Site_Commit

CADiZ ensures that the basic type rules have been correctly followed.

The function *Old_Value* is updated by the functional override operator in which the overriding function is a partial function defined as a set of maplets. This is a correct use of the functional override operator, therefore the function *Old_Value* retains its total functional properties.

The sets *Write_Lock* and *Read_Lock* are updated by removing members, hence if the restrictions imposed by the predicate in the schema Site_Data were valid for their before values, they must also be valid for their after values. Therefore, the invariants of the schema Site_Commit have not been violated.

Preconditions for the Schema Site_Commit

The schema Pre_Site_Commit_Simple defines the simplified preconditions, which are verified by expanding the schema Simplified_5_5.

Pre_Site_Commit_Simple

Site_Data

In_Ex? : Op

$\exists$ n_site : Trans_Num • In_Ex? = end (commit, n_site)

Simplified_5_5 $\triangleq$ pre Site_Commit $\Leftrightarrow$ Pre_Site_Commit_Simple

Finally, the **abort** operation is very similarly modelled by the schema Site_Abort.

Site_Abort

Site_Change

In_Ex? : Op

Existentially quantifying components

$\exists$ p_site : Operator; n_site : Trans_Num |

In_Ex? = end (p_site, n_site) $\wedge$ p_site = abort •

New_Value' =

New_Value $\oplus$

{o1 : Op; op1 : Operator; num1 : Trans_Num;

obj1 : Physical_Object; v1 : VALUE |

o1 = access (((op1, num1), obj1), v1) $\wedge$

o1 $\in$ Write_Lock $\wedge$ num1 = n_site •

obj1 $\mapsto$ Old_Value obj1 } $\wedge$

Read_Lock' =

Read_Lock \

{o1 : Op; op1 : Operator; num1 : Trans_Num;

obj1 : Physical_Object; v1 : VALUE |

o1 = access (((op1, num1), obj1), v1) $\wedge$ num1 = n_site •

o1 } $\wedge$

Write_Lock' =

Write_Lock \

{o1 : Op; op1 : Operator; num1 : Trans_Num;

obj1 : Physical_Object; v1 : VALUE |

o1 = access (((op1, num1), obj1), v1) $\wedge$ num1 = n_site •

o1 }

No change in the other declarations in Site_Data

$$Op\_Q' = Op\_Q$$
$$Old\_Value' = Old\_Value$$

The difference between the schemas Site_Abort and Site_Commit is that in the former schema the current values of the data objects used by the operations in the transactions and in the latter schema the values held by the data objects are restored to their previous values.

Note that, for simplicity, the schema Site_Abort does not address the question whether some other value should be used instead of its previous value because other transactions, which are not aborted, also change the data object.

Proof Obligation for the Schema Site_Abort

The proof sketch is identical to that for the schema Site_Commit, except that the function *New_Value* is updated instead of the function *Old_Value*.

Preconditions for the Schema Site_Abort

The following schema gives the simplified preconditions for the schema Site_Abort.

Pre_Site_Abort_Simple

$$Site\_Data$$
$$In\_Ex? : Op$$

$$\exists n\_site : Trans\_Num \bullet In\_Ex? = end\ (abort, n\_site)$$

The schema Pre_Site_Abort_Simple defines the simplified preconditions.

$$\text{Simplified\_5\_6} \vdash \text{pre Site\_Abort} \Leftrightarrow \text{Pre\_Site\_Abort\_Simple}$$

The above schema specifies the correctness condition for the simplified preconditions.

The site operation schemas are combined to form the schema Site_Op_Execution below.

$$\text{Site\_Op\_Execution} \mathrel{\hat{=}} \text{Site\_Read} \vee \text{Site\_Write} \vee \text{Site\_Commit} \vee \text{Site\_Abort}$$

The preconditions of the schema Site_Op_Execution are given by the following schema.

Pre_Site_Op_Execution_Simple

Site_Data

In_Ex? : Op

∃ b_site : Physical_Object; n_site : Trans_Num;

v_site : VALUE •

In_Ex? = access (((read, n_site), b_site), v_site) ∧

home = second b_site

read operation

∨

In_Ex? = access (((write, n_site), b_site), v_site) ∧

home = second b_site

write operation

∨

In_Ex? = end (commit, n_site)

commit operation

∨

In_Ex? = end (abort, n_site)

abort operation

This schema indicates that the schema Site_Op_Execution specifies the behaviour of the site execution schema for all values of the physical operations bound to the input variable *In_Ex?* that refer to the site with the value of *home*. Note that only one operation can be performed at any change of state.

5.4 Management Operations

This section defines the schemas that model the management aspects of the replicated database system. The schemas presented in this section fall into the four categories:

1 The invariant schemas, Section 5.4.1.

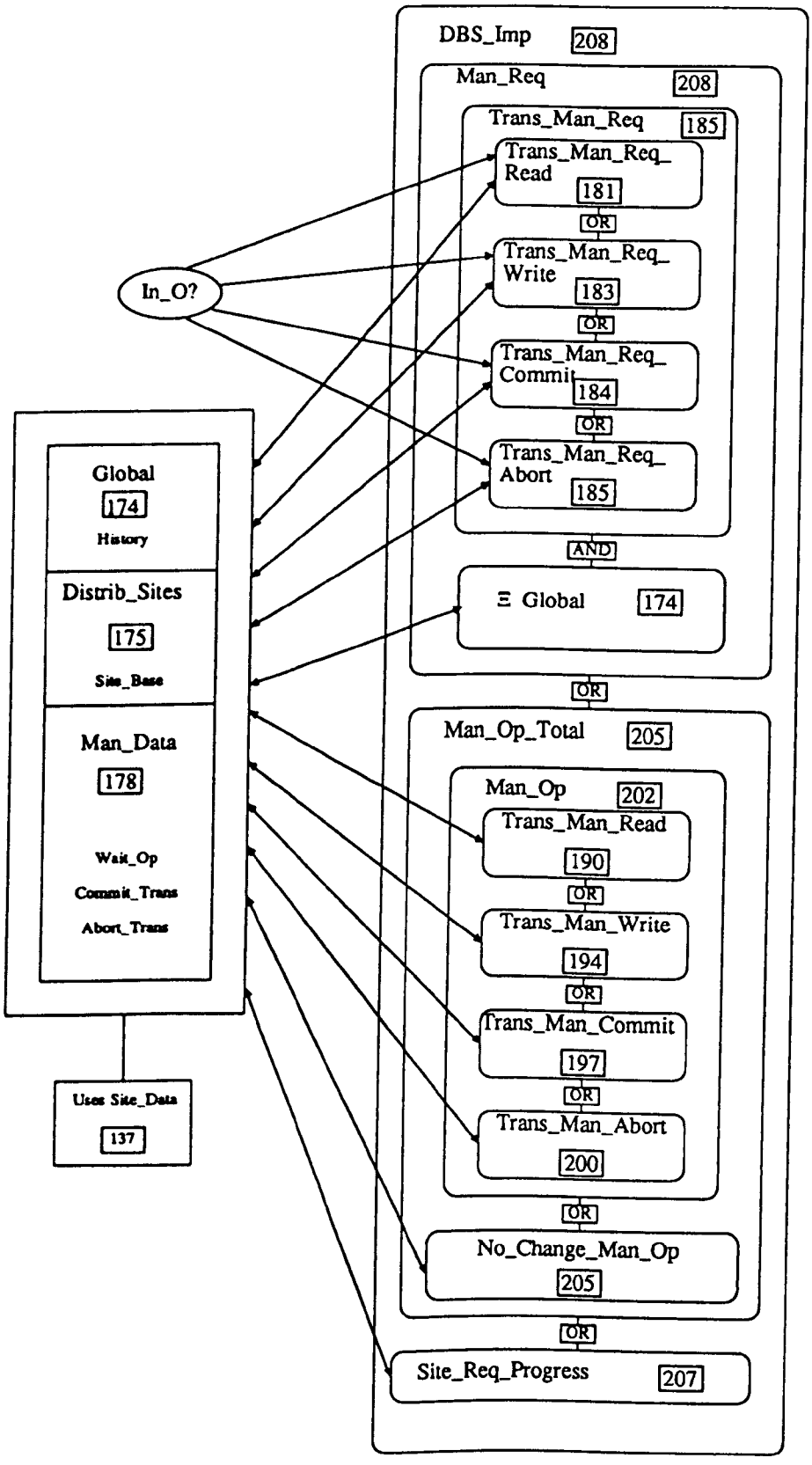
2 The schemas that describe the behaviour of the implementation in response to input logical operations, Section 5.4.2.

3 The schemas that describe the behaviour of the implementation executing the operations, Section 5.4.3.

4 The schema that describes the progress of all the site requests, Section 5.4.4.

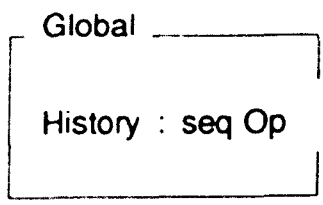
Figure 5.7 illustrates the main interactions between the state schemas and the management operation schemas.

Figure 5.7 Main Interactions between Management Schemas

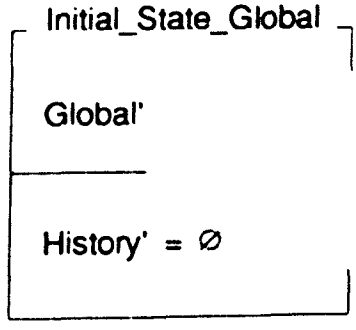


5.4.1 Invariants for Management Schemas

The schemas in the first category of management schemas include the schema Global below which defines a function that accumulates the execution of operations represented by the model of a replicated database system.



An initial state for the schema Global is defined as the following schema.



Proof Obligation for the Initial State

The empty sequence is a valid binding for the sequence *History* and hence a valid initial state must exist.

The schema Global is used only by the schemas that define operations that appear in the historical record.

The management of transactions uses logical operations as inputs and generates physical operations for the site schemas. The device of shared variables is used to exchange data between schemas instead of using input and output variables to exchange data. This allows schemas to be combined without using the pipe operator which is, although recognised by CADiZ, is not defined in the standard Z notation [Spiv89A].

$$
\begin{aligned}
&\mathrm{Logical\_Op} ::= \\
&\quad \mathrm{l\_access} \ll ((((\{\mathrm{read}, \mathrm{write}\} \times \mathrm{Trans\_Num}) \times \\
&\qquad\qquad \mathrm{LOGICAL\_OBJECT}) \times \mathrm{VALUE}) \gg \mid \\
&\quad \mathrm{l\_end} \ll (\{\mathrm{commit}, \mathrm{abort}\} \times \mathrm{Trans\_Num}) \gg
\end{aligned}
$$

The free type definition for Logical_Op is identical to the definition of Op except for using logical data objects instead of physical data objects.

Proof Obligation for the Free Type Definition of Logical_Op

Both branches are non recursive, hence the definition of Logical_Op is consistent.

The schema Distrib_Sites represents the concept of a number of sites operating independently, but all described by the same schema. Schema variables can be used instead of the function *Site_Base*, but this necessitates being explicit about the number of sites.

Distrib_Sites

$$\mathrm{Site\_Base} : \mathrm{Site} \rightarrow \mathrm{Site\_Data}$$

$$\forall s : \mathrm{Site} \bullet (\mathrm{Site\_Base}\ s).\mathrm{home} = s$$

$$\mathrm{Site} \neq \varnothing$$

One of the invariants of the schema Distrib_Sites is that the type Site is a non empty set, which represents the requirement that there is at least one site that has a copy of the database.

The function *Site_Base* provides a mapping from the type Site to the schema type Site_Data. The result of applying the function *Site_Base* to a value of a site is a particular binding of a schema such that the *home* component is equal to the value of the site used in the mapping.

The concept of bindings is difficult to understand in the context of abstract types. Some introductory books on the Z notation avoid using bindings [Hayes87, Dillr90, Craig91, Pottr91], but it is used in the book *The Z Notation: A Reference Manual* [Spiv89A] and is referred to in the semantics of the Z notation [ZipBS91]. The interpretation of bindings here is that a binding provides a mapping between an identifier and an element of the type of that identifier. For instance, a possible binding of an identifier *i* that represents natural numbers is $\{i \mapsto 3\}$.

Schemas can be used as types in the Z notation and elements of a schema type are given by the bindings of that schema [Spiv89A, Wood89B]. A binding provides a means of naming an element (or mathematical object) that has the schema type. Bindings are formed in the Z notation by using the θ operator, for example the expression:

θ *Initial_Site_State*

is a binding with

( ) as the value of the component *Op_Q'*

$\varnothing$ as the value of the component *Read_Lock'*

$\varnothing$ as the value of the component *Write_Lock'*

$\{d \mapsto$ Initial$\}$ for all physical objects for *Old_Value'*

$\{d \mapsto$ Initial$\}$ for all physical objects for *New_Value'*

The value bound to the variable *home'* is not specified by the schema Initial_Site_State.

The binding θ *Initial_Site_State* is an element of the type Initial_Site_State which is the same type as Site_Data.

Note that, the names of the components in the binding must be in scope at the point it is used.

An initial state for the schema Distrib_Sites is defined as the following schema.

Init_DS

Distrib_Sites'

Site_Data'

$\forall$ s : Site •

Site_Base' s = θ Initial_Site_State ∧

(Site_Base' s).home = s

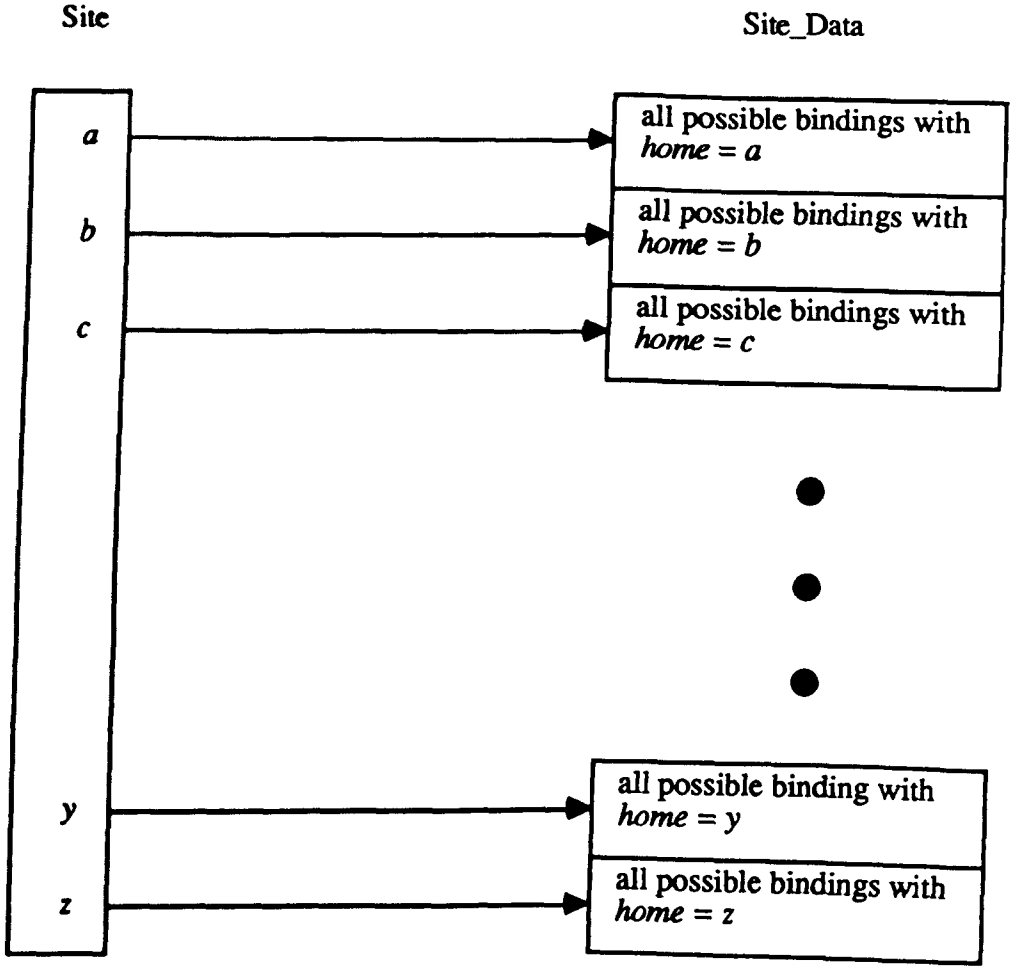
The schema Init_DS initialises each element in the mappings of *Site_Base* to the initial state of the the schema Site_Data, with the component *home* bound to a particular site value. The schema inclusion of Site_Data' is necessary to declare the variables used by the bindings given by the θ operator.

Proof Obligation for the Initial State

The binding given by Initial_Site_State is of the correct type and a value does exist that can be bound to each member of the set *Site*, which is syntactically equivalent to the given set LETTER. The component *home* can be bound to each member of the set *Site*. Should the given set LETTER be empty, then the predicate for the schema Distrib_Sites is false, hence cannot occur.

The purpose of the function *Site_Base* is to set up subsets of bindings of the schema type Site_Data to represent the bindings that are associated with each site. Figure 5.8 illustrates the mapping provided by the function *Site_Base*. The elements of the type Site_Data that refer to site *s* have the *home* component bound to the value *s*, similarly for other site values. The purpose of this partitioning of elements is to represent the concurrent activities of the sites by defining independent groups of states. An alternative interpretation of this is that the state is partitioned into a group of objects and the component *home* is the identity of each object.

Figure 5.8 Mapping of the Function Site_Data



The state of the management schemas is expressed in the following schema which defines the final state variables of the implementation.

Man_Data

Wait_Op : ℙ Logical_Op

Commit_Trans : ℙ Logical_Op

Abort_Trans : ℙ Logical_Op

All three sets *Wait_Op*, *Commit_Trans* and *Abort_Trans* are disjoint because only read

and write operations are members of *Wait_Op*, commit operations are members of *Commit_Trans*, and abort operations members of *Abort_Trans*. This can be specified as an invariant of the schema Man_Data, but it has no benefits because of the constructive style of the operations schemas.

An initial state for the schema Man_Data is defined by the schema Initial_Man_Data below.

Initial_Man_Data

Man_Data'

Wait_Op' = ∅

Commit_Trans' = ∅

Abort_Trans' = ∅

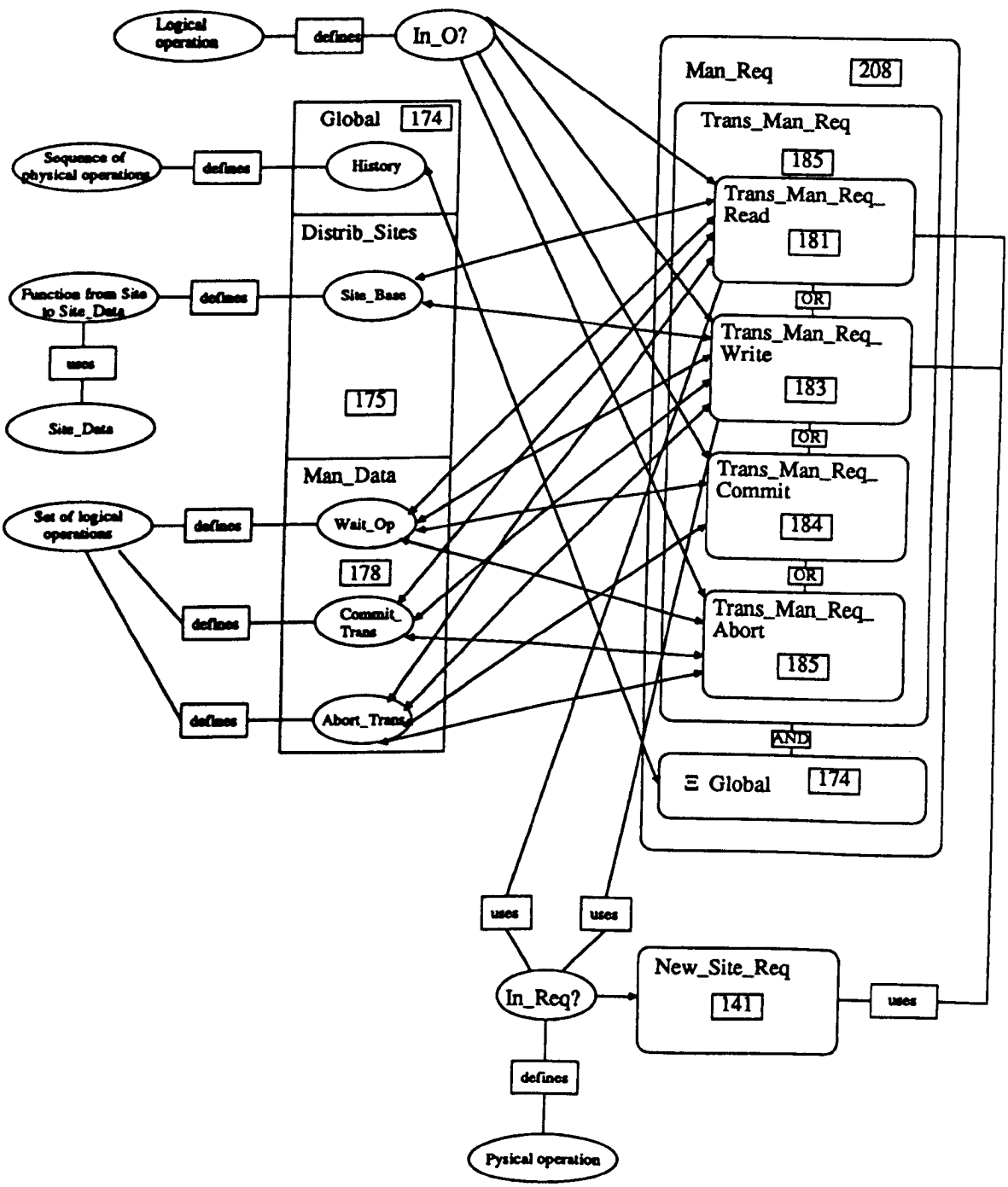
Proof Obligation for the Initial State

The empty set is an obvious valid state for all three sets *Wait_Op*, *Commit_Trans* and *Abort_Trans*.

5.4.2 Response to Logical Operation Requests

Figure 5.9 illustrates the interactions between the schemas for management requests to operations and the management state schemas.

Figure 5.9 Management Response to Requests



The definitions of the second category of management schemas include the request schemas. The first operation considered is the logical operation defined by the schema Trans_Man_Req_Read below.

Trans_Man_Req_Read

ΔDistrib_Sites

ΔMan_Data

In_O? : Logical_Op

$\exists$ p_man : Operator; n_man : Trans_Num;
lb_man : LOGICAL_OBJECT; v_man : VALUE |
In_O? = l_access (((p_man, n_man), lb_man), v_man) $\wedge$
p_man = read •
($\exists_1$ s : Site; op_man : Op |
op_man =
access (((p_man, n_man), (lb_man, s)), v_man) •
Site_Base' =
Site_Base $\oplus$
{s $\mapsto$
(μNew_Site_Req |
θSite_Data = Site_Base s $\wedge$ In_Req? = op_man •
θSite_Data')}) $\wedge$ Wait_Op' = Wait_Op $\cup$ {In_O?}

No change to other management data variables

Commit_Trans' = Commit_Trans

Abort_Trans' = Abort_Trans

The schema Trans_Man_Req_Read models the change of state that occurs as a consequence of receiving an input operation in the form of *In_O?*. Note, as was mentioned earlier, that this schema assumes that the read operation already has the value that is read

from the database.

The appropriate binding for the schema Site_Data is selected by using the function *Site_Base* within the μ operator that constructs a new unique binding of the schema type Site_Data.

The θ operator equates a binding of the schema Site_Data from the mapping provided by the function *Site_Base*. The local binding in the μ operator of the components of the schema Site_Data are equal to those of the binding of the target of the function *Site_Base* for the source value of *s*. It is this binding of the schema Site_Data that is used to update the data types in the site schemas by adding a physical read operator to exactly one site.

The only change of state caused by this operation is adding a new logical operation to the set *Wait_Op*.

Proof Obligation for the Schema Trans_Man_Req_Read

The management data components that are changed by the schema Trans_Man_Req_Read are the set *Wait_Op* and the function *Site_Base*. The type rules for the set *Wait_Op* are applied correctly, hence the invariant of the schema Man_Data is maintained.

The function *Site_Base* is updated using functional override, this ensures that the total functional characteristics are retained for *Site_Base*. The schema New_Site_Req is used to change the mapping for site *s* in the function *Site_Base*, this ensures that the component *home* in the schema Site_Data is not changed, hence maintaining the invariant of the schema Distrib_Sites.

The preconditions are not simplified for the single schema Trans_Man_Req_Read, instead the preconditions are given later for the disjunction of this and other request operation schemas defined below.

Trans_Man_Req_Write

ΔDistrib_Sites

ΔMan_Data

In_O? : Logical_Op

∃ p_man : Operator; n_man : Trans_Num;
lb_man : LOGICAL_OBJECT; v_man : VALUE |
In_O? = I_access (((p_man, n_man), lb_man), v_man) ∧
p_man = write •
Site_Base' =
Site_Base ⊕
{s : Site; op_man : Op |
op_man =
access (((p_man, n_man), (lb_man, s)), v_man) •
s ↦
(μNew_Site_Req |
θSite_Data = Site_Base s ∧ In_Req? = op_man •
θSite_Data')} ∧ Wait_Op' = Wait_Op ∪ {In_O?}

No other changes to management data variables

Commit_Trans' = Commit_Trans

Abort_Trans' = Abort_Trans

The schema Trans_Man_Req_Write specifies the change of state for a logical write operation received by the schema as an input. In this case, physical write operations are requested for all the sites, i.e. a write all procedure is used.

Proof Obligation for the Schema Trans_Man_Req_Write

The management data components that are changed by the schema Trans_Man_Req_Write are the set *Wait_Op* and the function *Site_Base*. The type

rules for the set *Wait_Op* are applied correctly, hence the invariant of the schema Man_Data is maintained.

The function *Site_Base* is updated using functional override, this ensures that the total functional characteristics are retained for *Site_Base*. The schema New_Site_Req is used to change the mapping for site *s* in the function *Site_Base*, this ensures that the component *home* in the schema Site_Data is not changed, hence maintaining the invariant of the schema Distrib_Sites.

The schema Trans_Man_Req_Commit receives an input of the type of a commit operation and adds the logical operation to the set of committed transactions.

Trans_Man_Req_Commit

ΔMan_Data

In_O? : Logical_Op

$\exists$ p_man : Operator; n_man : Trans_Num |

In_O? = l_end (p_man, n_man) $\wedge$ p_man = commit •

Commit_Trans' = Commit_Trans $\cup$ {In_O?}

No other changes to management data

Wait_Op' = Wait_Op

Abort_Trans' = Abort_Trans

Proof Obligation for the Schema Trans_Man_Req_Commit

The only change to the management data is to the set *Commit_Trans* in the schema Man_Data. Since the type rules are obeyed, the invariants are not violated.

Trans_Man_Req_Abort

ΔMan_Data

In_O? : Logical_Op

$\exists$ p_man : Operator; n_man : Trans_Num |

In_O? = l_end (p_man, n_man) $\wedge$ p_man = abort •

Abort_Trans' = Abort_Trans $\cup$ {In_O?}

No other changes to management data

Wait_Op' = Wait_Op

Commit_Trans' = Commit_Trans

Trans_Man_Req_Abort receives an input logical abort operation and adds the logical operation to the set for aborted transactions.

Proof Obligation for the Schema Trans_Man_Req_Abort

The only change to the management data is to the set *Abort_Trans* in the schema Man_Data. Since the type rules are obeyed, the invariants are not violated.

The schema Trans_Man_Req is defined as the disjunction of the four schemas for each operation.

Trans_Man_Req $\hat{=}$

Trans_Man_Req_Read $\vee$ Trans_Man_Req_Write $\vee$

Trans_Man_Req_Commit $\vee$ Trans_Man_Req_Abort

Note that only one operation is possible for each change of state.

The data declarations in the management request schemas are not the same. In particular the commit and abort management request schemas do not use the schema Distrib_Sites

that is included in the read and write management request schemas. This means that each schema in the disjunction does not specify all the components of the composite state. Strictly, those variables not included in the component schema declarations should be defined explicitly as being unaffected by the operation defined by the schema. This is achieved by declaring a schema that indicates that there is no change of state for these variables in the conjunction of the commit and abort schemas.

$$\text{Trans\_Man\_Req\_Commit1} \triangleq \text{Trans\_Man\_Req\_Commit} \wedge \Xi \text{Distrib\_Sites}$$

and

$$\text{Trans\_Man\_Req\_Abort1} \triangleq \text{Trans\_Man\_Req\_Abort} \wedge \Xi \text{Distrib\_Sites}$$

These new schemas can be used to give new versions of the schema Trans_Man_Req, but to simplify the presentation it is not done here.

Preconditions for the Schema Trans_Man_Req

The preconditions for the schema Trans_Man_Req are simplified to those given by the schema Pre_Man_Req_Simple below.

Pre_Man_Req_Simple

Distrib_Sites

Man_Data

In_O? : Logical_Op

$\exists$ n_man : Trans_Num; lb_man : LOGICAL_OBJECT;

pb_man : Physical_Object; v_man : VALUE •

In_O? = l_access (((read, n_man), lb_man), v_man)

read operation

$\vee$

In_O? = l_access (((write, n_man), lb_man), v_man)

write operation

$\vee$

In_O? = l_end (commit, n_man)

commit operation

$\vee$

In_O? = l_end (abort, n_man)

abort operation

The correctness of the simplified preconditions is expressed in the schema below.

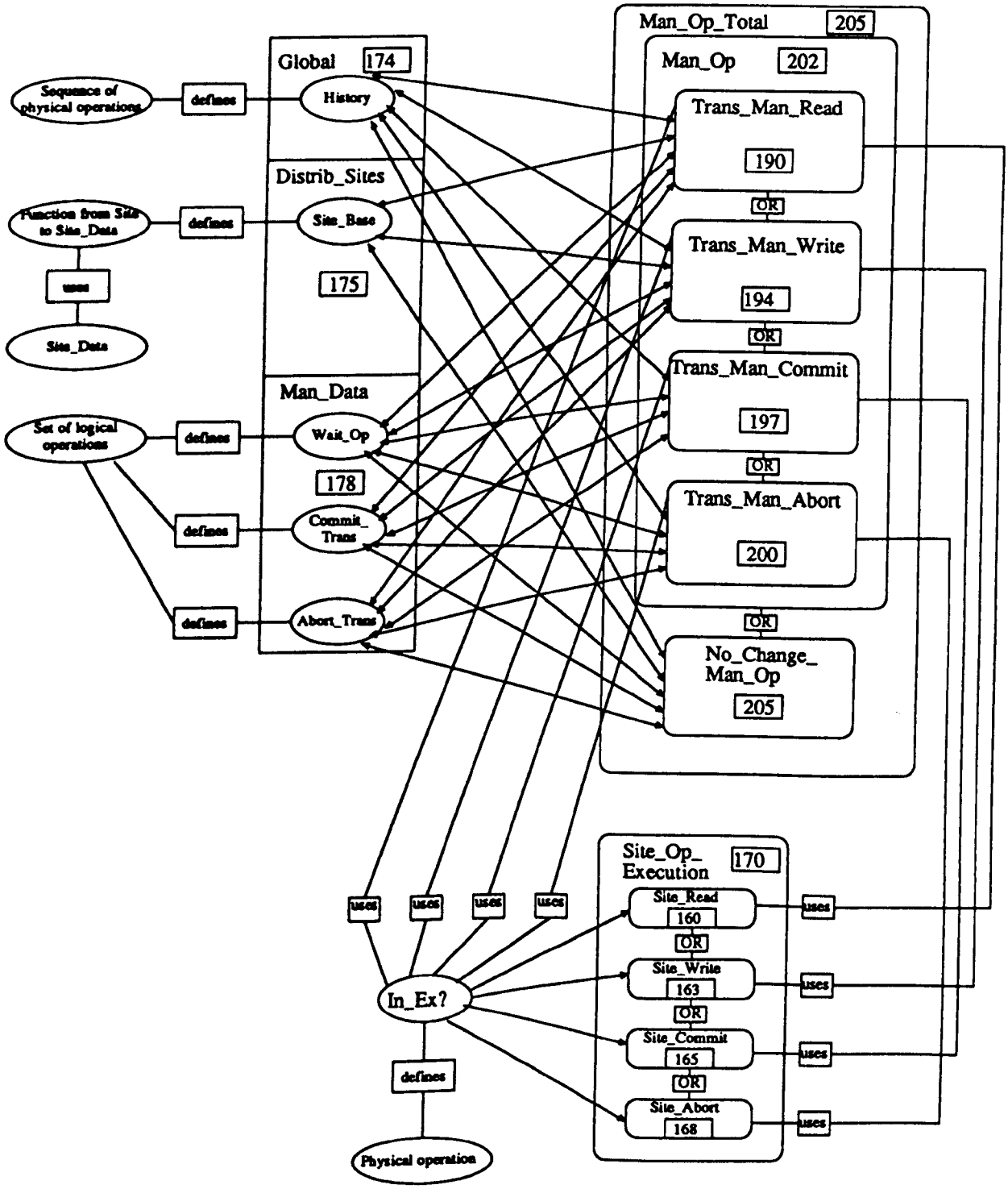
$$\text{Simplified\_5\_7} \triangleq \text{pre Trans\_Man\_Req} \Leftrightarrow \text{Pre\_Man\_Req\_Simple}$$

The schema Trans_Man_Req models the queueing of requests for physical access operations to be performed by the sites in the distributed database system. The site schemas grant these requests when there are no conflicting operations.

5.4.3 Execution of Logical Operations

Figure 5.10 illustrates the interactions between the management execution schemas and the management state schemas.

Figure 5.10 Schemas for Management Execution



The following four schemas form the management execution category of schemas. These schemas respond to the grants generated by the site schemas by constructing the physical operations that are executed at the sites and forming the historical record of operations.

The following schema Trans_Man_Read specifies the management read operation. The schema Site_Read to define the change of state instead of the schema Site_Op_Execution. This is justified because any expansion of the schema will simplify to using the schema Site_Read.

Trans_Man_Read

ΔGlobal

ΔDistrib_Sites

ΔMan_Data

$\exists$ p_man : Operator; n_man : Trans_Num;

lb_man : LOGICAL_OBJECT; v_man : VALUE;

lr_man : Logical_Op |

lr_man = l_access (((p_man, n_man), lb_man), v_man) $\wedge$

p_man = read $\wedge$ lr_man $\in$ Wait_Op $\bullet$

$\exists_1$ s : Site; op_man : Op |

op_man =

access (((p_man, n_man), (lb_man, s)), v_man) $\bullet$

op_man $\in$ (Site_Base s) . Read_Lock \ ran History $\wedge$

Site_Base' =

Site_Base $\oplus$

{s $\mapsto$

(μSite_Read |

θSite_Data = Site_Base s $\wedge$ In_Ex? = op_man $\bullet$

θSite_Data')} $\wedge$ Wait_Op' = Wait_Op \ {lr_man} $\wedge$

History' = History $\frown$ ⟨op_man⟩

No other changes to the management data variables

Commit_Trans' = Commit_Trans

Abort_Trans' = Abort_Trans

The schema Trans_Man_Read selects a member from the set *Wait_Op* and existentially quantifies the components in the free type definition of a Logical_Op. The additional preconditions of this schema include both that there exists a single site that has granted

permission for a read operation to occur on the physical object stored at a site and that the operation has not be executed already (as denoted by the set difference between the set *Read_Lock* and the operations in the sequence *History*).

The schema Site_Data is given the appropriate binding for the site in question by the function *Site_Base*.

The change of state is indicated by changes to the particular binding of the schema Site_Data, and updating of the set *Wait_Op* and sequence *History*, all within the scope of the existential quantifier with the bound variable *s*.

In the schema Trans_Man_Read, the preconditions and postconditions are combined because of the scope rules of predicate calculus makes this approach simpler than separating the preconditions.

The changes of the variables declared by the schema Site_Data are responded to by the site schemas for the execution of operations.

Proof Obligation for the Schema Trans_Man_Read

The schema Trans_Man_Read changes the set *Wait_Op*, the sequence *History* and the function *Site_Base*. The only invariants for the set *Wait_Op* and the sequence *History* relate to their types. Since the type rules are applied correctly, the invariants are not violated.

The changes to the function *Site_Base* use the functional override operation such that a single element is overridden. This means that the total functional properties of *Site_Base* are not violated. The schema Site_Read does not change the component *home* in the schema Site_Data, hence the invariant for the schema Distrib_Sites is maintained.

Preconditions for the Schema Trans_Man_Read

The schema Pre_Man_Read_Simple gives the simplified preconditions for the schema Trans_Man_Read.

Pre_Man_Read_Simple

Global

Distrib_Sites

Man_Data

$\exists$ n_man : Trans_Num; lb_man : LOGICAL_OBJECT;
v_man : VALUE; lr_man : Logical_Op |
lr_man = l_access (((read, n_man), lb_man), v_man) $\wedge$
lr_man $\in$ Wait_Op •
$\exists$ s : Site •
access (((read, n_man), (lb_man, s)), v_man) $\in$
(Site_Base s). Read_Lock \ ran History

The following schema is verified to ensure that the preconditions are correct.

$$\text{Simplified\_5\_8} \triangleq \text{pre Trans\_Man\_Read} \Leftrightarrow \text{Pre\_Man\_Read\_Simple}$$

The schema Trans_Man_Write uses a generic operator for mapping a set of elements to a sequence of the same elements in any order. The relation between sets and sequences is given below.

[X]

ordering : $\mathbb{P}$ X $\leftrightarrow$ seq X

$\forall$ set_x : $\mathbb{P}$ X; seq_x : seq X | ordering set_x = seq_x •
ran seq_x = set_x

The schema Trans_Man_Write models the effects of all sites simultaneously agreeing to

write operations being executed on the physical data objects that correspond to the logical data object.

Trans_Man_Write

ΔGlobal

ΔDistrib_Sites

ΔMan_Data

$\exists$ p_man : Operator; n_man : Trans_Num;
lb_man : LOGICAL_OBJECT; v_man : VALUE;
lr_man : Logical_Op |
lr_man = l_access (((p_man, n_man), lb_man), v_man) $\wedge$
p_man = write $\wedge$ lr_man $\in$ Wait_Op •
($\forall$ s : Site; op_man : Op |
op_man =
access (((p_man, n_man), (lb_man, s)), v_man) •
op_man $\in$ (Site_Base s) . Write_Lock \ ran History) $\wedge$
Site_Base' =
Site_Base $\oplus$
{s : Site; op_man : Op |
op_man =
access (((p_man, n_man), (lb_man, s)), v_man) •
s $\mapsto$
(μSite_Write |
θSite_Data = Site_Base s $\wedge$ In_Ex? = op_man •
θSite_Data')} $\wedge$
History' =
History $\frown$
ordering
{s : Site; op_man : Op |

$$op\_man = access(((p\_man, n\_man), (lb\_man, s)), v\_man) \bullet op\_man\} \land Wait\_Op' = Wait\_Op \setminus \{lr\_man\}$$

No other changes to the management data variables

$$Commit\_Trans' = Commit\_Trans$$

$$Abort\_Trans' = Abort\_Trans$$

Proof Obligation for the Schema Trans_Man_Write

The schema Trans_Man_Write changes the set *Wait_Op*, the sequence *History* and the function *Site_Base*. The only invariants for the set *Wait_Op* and the sequence *History* relate to their types. Since the type rules are applied correctly, the invariants are not violated.

The changes to the function *Site_Base* use the functional override operation such that all the elements in its domain are overridden. This means that the total functional properties of *Site_Base* are not violated. The schema Site_Write does not change the component *home* in the schema Site_Data, hence the invariant for the schema Distrib_Sites is maintained.

Preconditions for the Schema Trans_Man_Write

The preconditions of the schema Trans_Man_Write include that the logical operation is in the set *Wait_Op* and that all the sites have granted permission for the associated physical write operations, as indicated by the condition of the different bindings of the set *Write_Lock* given by the function *Site_Base*.

Pre_Man_Write_Simple

Global

Distrib_Sites

Man_Data

$\exists$ n_man : Trans_Num; lb_man : LOGICAL_OBJECT;
v_man : VALUE; lr_man : Logical_Op |
lr_man = l_access (((write, n_man), lb_man), v_man) $\wedge$
lr_man $\in$ Wait_Op •
$\forall$ s : Site •
access (((write, n_man), (lb_man, s)), v_man) $\in$
(Site_Base s). Write_Lock \ ran History

The schema Pre_Man_Write_Simple defines the preconditions that are derived from the schema Trans_Man_Write. The simplification is verified by expanding the following schema.

Simplified_5_9 $\hat{=}$ pre Trans_Man_Write $\Leftrightarrow$ Pre_Man_Write_Simple

The schema Trans_Man_Commit below models the management functions associated with the execution of a commit operation.

Trans_Man_Commit

ΔGlobal

ΔDistrib_Sites

ΔMan_Data

$\exists$ lr_man : Logical_Op; n_man : Trans_Num; op_man : Op;

p_man : Operator |

lr_man = l_end (p_man, n_man) $\wedge$ p_man = commit $\wedge$

lr_man $\in$ Commit_Trans $\wedge$

$\neg$

($\exists$ l2 : Logical_Op; num2 : Trans_Num; op2 : Operator;

log_obj2 : LOGICAL_OBJECT; v2 : VALUE $\bullet$

l2 = l_access (((op2, num2), log_obj2), v2) $\wedge$

num2 = n_man $\wedge$ l2 $\in$ Wait_Op) $\bullet$

op_man = end (commit, n_man) $\wedge$

History' = History $\frown$ ⟨op_man⟩ $\wedge$

Commit_Trans' = Commit_Trans \ {lr_man} $\wedge$

Site_Base' =

Site_Base $\oplus$

{s : Site $\bullet$

s $\mapsto$

(μSite_Commit |

θSite_Data = Site_Base s $\wedge$ ln_Ex? = op_man $\bullet$

θSite_Data')}

No other changes to management data

Wait_Op' = Wait_Op

Abort_Trans' = Abort_Trans

Proof Obligation for the Schema Trans_Man_Commit

The schema Trans_Man_Commit changes the set *Commit_Op*, the sequence *History* and the function *Site_Base*. The only invariants for the set *Commit_Op* and the sequence *History* relate to their types. Since the type rules are applied correctly, the invariants are not violated.

The changes to the function *Site_Base* use the functional override operation such that all the elements in its domain are overridden. This means that the total functional properties of *Site_Base* are not violated. The schema Site_Commit does not change the component *home* in the schema Site_Data , hence the invariant for the schema Distrib_Sites is maintained.

Preconditions for the Schema Trans_Man_Commit

The preconditions and postconditions are combined in the above schema because both sets of conditions use the same quantified variables. The first existential quantifier extracts a member from the set *Commit_Trans*. This set contains all the commit operations waiting to be executed. The second existential quantifier ensures that there are no logical operations with the same transaction number waiting to be executed, as indicated by membership of the set *Wait_Op*. A commit operation is sent to all sites for each transaction, whether that site has been involved or not.

The schema Pre_Man_Commit_Simple below specifies the simplified preconditions for the schema Trans_Man_Commit.

Pre_Man_Commit_Simple

Global

Distrib_Sites

Man_Data

$$
\begin{aligned}
&\exists\, lr\_man : Logical\_Op;\ n\_man : Trans\_Num \mid \\
&\quad lr\_man = l\_end\ (commit, n\_man)\ \bullet \\
&\qquad lr\_man \in Commit\_Trans\ \wedge \\
&\qquad \neg \\
&\qquad (\exists\, l2 : Logical\_Op;\ num2 : Trans\_Num;\ op2 : Operator; \\
&\qquad\quad log\_obj2 : LOGICAL\_OBJECT;\ v2 : VALUE\ \bullet \\
&\qquad\qquad l2 = l\_access\ (((op2, num2), log\_obj2), v2)\ \wedge \\
&\qquad\qquad num2 = n\_man\ \wedge\ l2 \in Wait\_Op)
\end{aligned}
$$

The schema below indicates the condition for the simplified preconditions to be correct.

$$Simplified\_5\_10 \mathrel{\widehat{=}} pre\ Trans\_Man\_Commit \Leftrightarrow Pre\_Man\_Commit\_Simple$$

The schema Trans_Man_Abort below models the functions associated with the execution of an abort operation, which are very similar to those functions associated with a commit operation.

Trans_Man_Abort

ΔGlobal

ΔDistrib_Sites

ΔMan_Data

∃ lr_man : Logical_Op; p_man : Operator;

n_man : Trans_Num; op_man : Op |

lr_man = l_end (p_man, n_man) ∧ p_man = abort ∧

lr_man ∈ Abort_Trans ∧

¬

(∃ l2 : Logical_Op; num2 : Trans_Num; op2 : Operator;

log_obj2 : LOGICAL_OBJECT; v2 : VALUE •

l2 = l_access (((op2, num2), log_obj2), v2) ∧

num2 = n_man ∧ l2 ∈ Wait_Op) •

op_man = end (abort, n_man) ∧

History' = History ⌢ ⟨op_man⟩ ∧

Abort_Trans' = Abort_Trans \ {lr_man} ∧

Site_Base' =

Site_Base ⊕

{s : Site •

s ↦

(μSite_Abort |

θSite_Data = Site_Base s ∧ In_Ex? = op_man •

θSite_Data')}

No other changes to management data

Wait_Op' = Wait_Op

Commit_Trans' = Commit_Trans

Proof Obligation for the Schema Trans_Man_Abort

The proof sketch is identical to that for the schema Trans_Man_Commit except that the set *Abort_Trans* is changed and the schema Site_Abort is used to update the bindings in the function *Site_Base*.

Preconditions for the Schema Trans_Man_Abort

The preconditions are simplified to give the following schema.

Pre_Man_Abort_Simple

Global
Distrib_Sites
Man_Data

$$\exists\, lr\_man : Logical\_Op;\ n\_man : Trans\_Num \mid$$
$$lr\_man = l\_end\ (abort, n\_man) \bullet$$
$$lr\_man \in Abort\_Trans \wedge$$
$$\neg$$
$$(\exists\, l2 : Logical\_Op;\ num2 : Trans\_Num;\ op2 : Operator;$$
$$log\_obj2 : LOGICAL\_OBJECT;\ v2 : VALUE \bullet$$
$$l2 = l\_access\ (((op2, num2), log\_obj2), v2) \wedge$$
$$num2 = n\_man \wedge l2 \in Wait\_Op)$$

The correctness of the simplified preconditions is expressed by the schema below.

$$Simplified\_5\_11 \triangleq pre\ Trans\_Man\_Abort \Leftrightarrow Pre\_Man\_Abort\_Simple$$

The schema Man_Op forms a disjunction of the management request operations. Note that all the schema terms are mutually exclusive because each term defines the postconditions uniquely.

$$Man\_Op \triangleq$$
$$Trans\_Man\_Read \lor Trans\_Man\_Write \lor$$
$$Trans\_Man\_Commit \lor Trans\_Man\_Abort$$

Only one logical operation is possible for each change of state in the schema Man_Op, however, a logical operation may involve multiple sites.

The preconditions of the schema Man_Op are given by the following schema.

Pre_Man_Simple

Global

Distrib_Sites

Man_Data

$\exists$ n_man : Trans_Num; lb_man : LOGICAL_OBJECT;
pb_man : Physical_Object; v_man : VALUE;
lr_man : Logical_Op •
lr_man = l_access (((read, n_man), lb_man), v_man) $\wedge$
lr_man $\in$ Wait_Op $\wedge$
($\exists_1$ s : Site •
access (((read, n_man), (lb_man, s)), v_man) $\in$
(Site_Base s) . Read_Lock \ ran History)
read operation
$\vee$
lr_man = l_access (((write, n_man), lb_man), v_man) $\wedge$
lr_man $\in$ Wait_Op $\wedge$
($\forall$ s : Site •
access (((write, n_man), (lb_man, s)), v_man) $\in$
(Site_Base s) . Write_Lock \ ran History)
write operation
$\vee$
lr_man $\in$ Commit_Trans $\wedge$
lr_man = l_end (commit, n_man) $\wedge$
$\neg$
($\exists$ l2 : Logical_Op; num2 : Trans_Num; op2 : Operator;

log_obj2 : LOGICAL_OBJECT; v2 : VALUE •

l2 = l_access (((op2, num2), log_obj2), v2) ∧

num2 = n_man ∧ l2 ∈ Wait_Op)

commit operation

∨

lr_man ∈ Abort_Trans ∧ lr_man = l_end (abort, n_man) ∧

¬

(∃ l2 : Logical_Op; num2 : Trans_Num; op2 : Operator;

log_obj2 : LOGICAL_OBJECT; v2 : VALUE •

l2 = l_access (((op2, num2), log_obj2), v2) ∧

num2 = n_man ∧ l2 ∈ Wait_Op)

abort operation

The schema Man_Op is not total over all possible state values. The states other than those that meet the precondition Pre_Man_Simple are given by the schema below.

Not_Pre_Man_Simple ≙ ¬ Pre_Man_Simple

This also includes the states that do not meet the invariants of the schema Pre_Man_Simple. The invariants are included in the following schema:

Not_Pre_Man_Simple1 ≙

¬ Pre_Man_Simple ∧ Global ∧ Distrib_Sites

However, the schema invariants can also be included by specifying that there is no change of state as in the schema below.

$$No\_Change\_Man\_Op \triangleq$$

$$\neg\ Pre\_Man\_Simple \wedge \Xi Global \wedge \Xi Distrib\_Sites$$

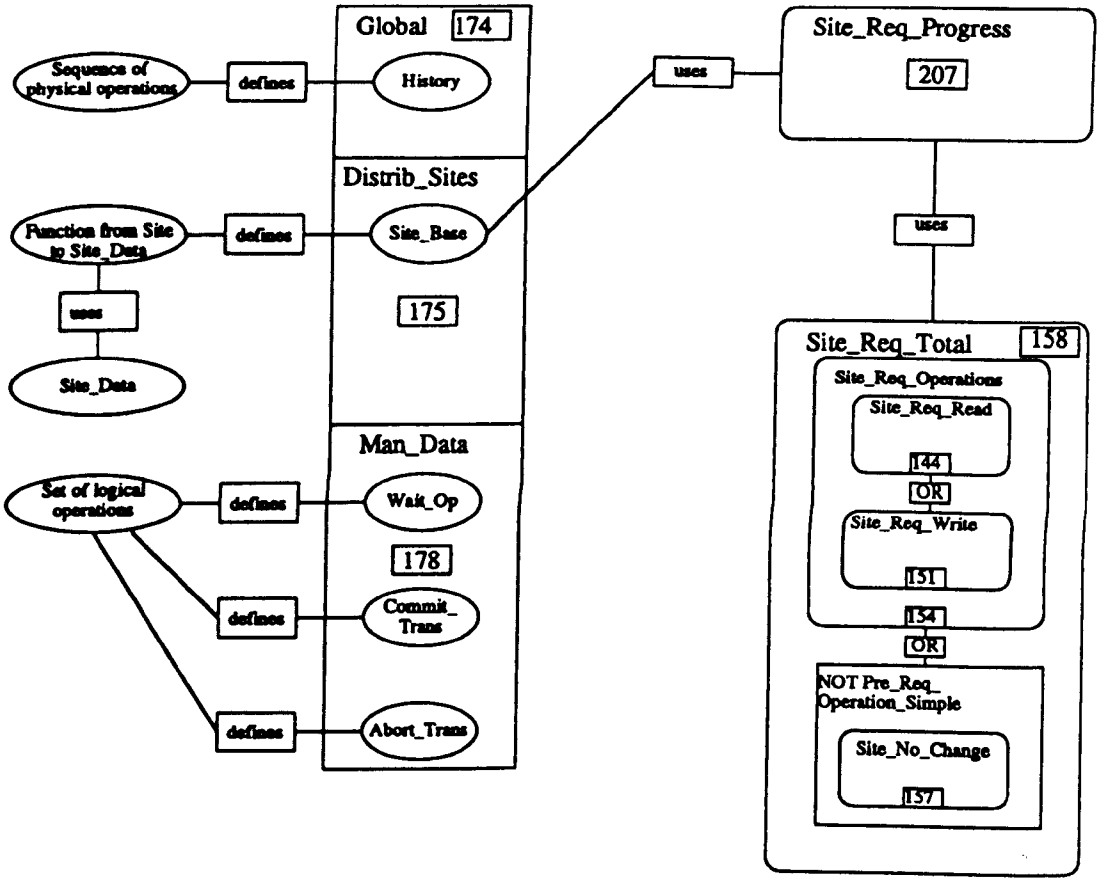
Giving the final total specification of the management execution operation in the form of the schema below.

$$Man\_Op\_Total \triangleq Man\_Op \vee No\_Change\_Man\_Op$$

5.4.4 Progress of Site Requests

Figure 5.11 illustrates he interactions between the schema for the progress of the site requests and the management state schemas.

Figure 5.11 Schema for Site Progress



The state changes brought about by the site request operations to all sites are represented by the schema Site_Req_Progress.

Site_Req_Progress

$$\Delta Distrib\_Sites$$
$$\Xi Man\_Data$$
$$\Xi Global$$

$$Site\_Base' = Site\_Base \oplus \{s : Site \bullet s \mapsto (\mu Site\_Req\_Total \mid \theta Site\_Data = Site\_Base\ s \bullet \theta Site\_Data')\}$$

Note that a number of sites can be affected in a single change of state.

The schema Site_Req_Progress includes the conditions that the variables in the schemas Man_Data and Global are not changed, hence prohibiting concurrent changes of site request actions and any other type of action.

Proof Obligation for the Schema Site_Req_Progress

The function *Site_Base* is changed by all elements in its domain being overridden by the results of the schema Site_Req_Total for that element being bound to the component *home* of the schema Site_Data. This can result in changes to the bindings in the range of *Site_Base*, however, the component *home* is never changed, thereby maintaining the invariant of the schema Distrib_Sites.

Preconditions for the Schema Site_Req_Progress

The only preconditions for the schema Site_Req_Progress are the correct types for the components in the management data. The schema Site_Req_Total is total for all the state values. Some state values result in changes to the components *Read_Lock* and *Write_Lock* for particular sites. Other state values do not cause any change to the bindings of *Site_Base*.

5.5 Complete Implementation

The schema Man_Req combines the transaction management request schema with the declaration of the history records since they are not changed by this management schema.

$$\text{Man\_Req} \triangleq \Xi\text{Global} \wedge \text{Trans\_Man\_Req}$$

The implementation of the replicated database system is defined as:

$$\text{DBS\_Imp} \triangleq \text{Man\_Req} \vee \text{Man\_Op\_Total} \vee \text{Site\_Req\_Progress}$$

Again, schema disjunction indicates the independent actions of each of the three schema terms. Because the preconditions of the schemas are not mutually exclusive there is a possibility that state changes are caused by more than one schema term. However, the postconditions ensure that the change of state is due to one schema only.

The interpretation of the functional behaviour represented by the schemas is summarised as follows. An input representing a logical operation is received by the schema Trans_Man_Req . The schema Trans_Man_Req updates the input variable to the schema New_Site_Req. The schema New_Site_Req in turn updates the sequence of operations for each site. The schema Site_Req_Progress monitors the sequence of operations for each site and updates the components *Read_Lock* and *Write_Lock* when possible. The schema Man_Op responds to the values of the components *Read_Lock* and *Write_Lock* to change the values bound to the input variable in the site operation execution

schemas included in Site_Op_Execution.

The verification condition applies to complete histories of the operations contained in the transactions. The operation schemas in the implementation apply to individual operations and, as a byproduct, construct a historical record of the operations. The historical record that is referred to in the verification is constructed from a sequence of operations whose behaviours are specified by the operation schemas in the implementation.

The schema DBS_Imp represents all the possible bindings that meet the conditions of the schema, hence are the effects of the operation schemas that constitute DBS_Imp.

The schema DBS_History below represents the states of the implementation that exist for complete histories.

$$DBS\_History \triangleq DBS\_Imp \wedge Complete\_History_{[History/History\_Rec]}$$

5.6 Proof Sketches of the Serializability of the Implementation

This section presents two informal proof sketches of the serializability of the transactions performed on the implementation of a replicated database system presented in this chapter. The first proof sketch verifies that all the transactions are one copy serializable by means of an induction on the length of the historical record of operations without explicit reference to the one copy serialization property that is expressed in the Z notation.

The second proof sketch uses both the specification of the one copy serialization property and the implementation to prove that all histories in the implementation can occur in the histories generated by the property schema.

Appendix C contains several example animations of the schemas contained in this chapter. Looking at the examples may make the proof sketches easier to follow.

5.6.1 Proof Sketch of One Copy Serializability

The proof is an induction proof based on the length of the sequence of operations bound to the variable *History*.

Base Case:

History = ⟨ ⟩

Since there are no operations, there cannot be any conflicts.

Induction step:

History = $h1$ *of length* m

Assume that a history, $h1$, of length m is serializable. A new history, $h1'$, of length $m + 1$ is formed by concatenating an operation $o1$, such that

$h1' = h1 \frown \langle o1 \rangle$

All possible replicated data objects histories are serializable histories is proved by case analysis on each of the four types of operators:

1 Read operation of physical object xa by transaction i.

Any conflict this operation has with previous operations is caused by either of the following two possibilities:

(i) The previous operation is a write operation to the same logical object x from a completed transaction with number j.

From the write all algorithm specified by the schema Trans_Man_Write, all sites locked a write request to the physical copies of x, including the physical object xa. The physical operations in the set *Write_Lock* are removed by the schemas Site_Commit or Site_Abort, indicating a completed transaction. Therefore the effects of this previous write must occur before the current read operation and an equivalent serial order of transactions is j occurs before i.

(ii) The previous operation is a write operation to the same logical object x from an

uncompleted transaction with number *j*.

This is impossible because all bindings given by the function *Site_Base* of the set *Write_Lock* will have a member which is a physical write operation to each copy of *x*, including *xa*, hence making the preconditions of the schema Site_Req_Read false.

2 Write operation of physical object *xa* by transaction *i*.

Any conflict this operation has with previous operations is caused by one of the following four possibilities:

(i) A previous operation is a write operation to the same logical object *x* from a completed transaction with number *j*.

Because of the write all algorithm, all copies of the logical object *x* have been updated, hence transaction *j* can be put in a serial order with transaction *i* such that transaction *j* occurs before transaction *i*.

(ii) A previous operation is a write operation to the same logical object *x* from an uncompleted transaction with number *j*.

This cannot occur because the members of *Write_Lock* include a physical operation to object *x*.

(iii) A previous operation is a read operation to the same logical object *x* from a completed transaction with number *j*.

The transaction *j* cannot have any conflicting operations with transaction *i* at the time of completion, hence an equivalent serial is possible with transaction *j* occurring before transaction *i*.

(iv) A previous operation is a read operation to the same logical object *x* from an uncompleted transaction with number *j*. There must exist some site that has a physical read operation as a member of the set *Read_Lock*. Thus causing the preconditions of the schema Site_Req_Write to be false and, as a consequence, also causing the preconditions of the schema Trans_Man_Write to be false, hence this combination of events cannot occur.

3 Commit operation

Since a commit operation can be performed at any time, provided if it is the last operation of any transaction, $h1'$ is a one copy serializable history because $h1$ is a one copy serializable history.

4 Abort operation

Identical reasoning applies to the abort operation as to the above commit operation.

5.6.2 Proof Sketch of the Behaviour meeting the Specification Property

Even with both the specification and the implementation written in the same notation, it is not a trivial task to relate a description of a specification to a description of an implementation because of the different levels of abstraction. However, in this case the specification is written in terms of a data type that is also in the implementation, i.e. that of a sequence of operations. This common data type allows direct comparisons to be made between the two descriptions.

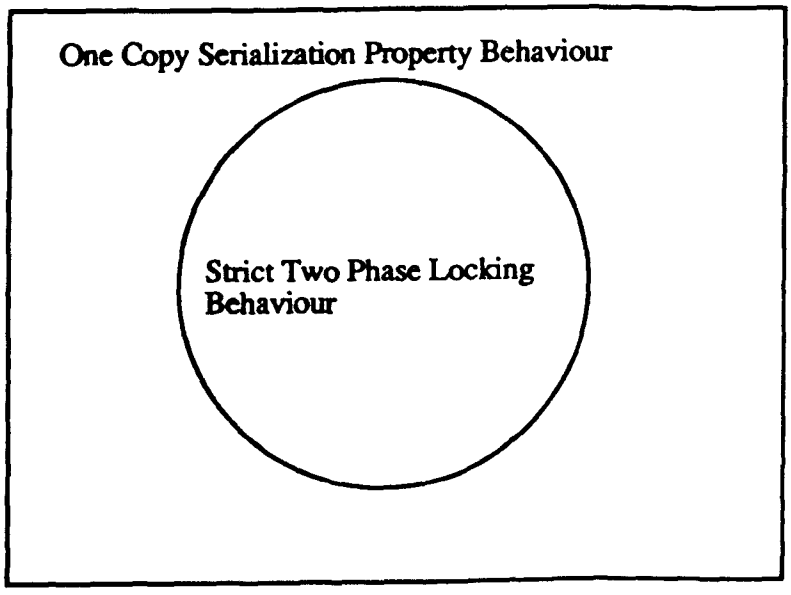
If different data types are used, then some relation (i.e. an abstraction relation) has to be given that specifies the rules for converting from one data type to the other.

The hypothesis is that each possible complete history under the replicated database schemas is also possible under the serializability schemas. The correctness criterion is of the form:

$$Implementation \Rightarrow Specification$$

This can be considered in terms of behaviours where, if an implementation can give rise to a certain behaviour, then that behaviour possesses the required property. Figure 5.12 represents this relationship as a Venn diagram, the set representing the behaviour of an implementation is a subset of the set representing the behaviour implied by the specification.

Figure 5.12 Inclusion Relationship for One Copy Serialization Property



In both the implementation and the specification in this study, the behaviour is represented as sequences of operations and the verification that the implementation of the replicated database system has the one copy serialization property is expressed as the following theorem for all complete histories:

$$DBS\_History \wedge Serializable\_History \vdash History \in Histories$$

This states that all complete histories of operations that are produced by the implementation are members of the set of all histories of operations that have the one copy serialization property.

This can be interpreted as, there can be histories of operations that possess the one copy serialization property other than those capable of being generated by the version of the strict two phase locking implementation, but all those possible by the implementation do have the one copy serialization property.

The proof strategy is to show that any complete history of operations that can occur in the implementation, is also possible in the specification.

Proof Sketch of Correctness

Consider an arbitrary history value, h, that records the operations that meet the implementation represented by the schema DBS_History. It must be shown possible to construct the same history value using the schema Complete_Serializable that represents the specification.

The following proof refers to the schemas that are used to construct the specification and it will be useful to refer to Figure 5.1 while reading this proof sketch.

The serialization property only applies to complete histories. The history value h satisfies the schema Complete_History that is part of the specification because the same schema is also used in the implementation. Therefore, all the transaction in h are terminated by either a commit operation or an abort operation.

The schema History_Invariant restricts the component *History_Rec*, which in this case has the binding of the value h. The component *Op_Q* in the schema Site_Data is defined to be a sequence of physical operations, this data structure is used by the schema Site_Req_Operations to ensure that the order in which physical operations are performed by each site is the same as the order in which the operations appear in the transactions. The only precedence relation in the implementation is between operations within transactions and that there are no other restrictions placed on the operations. This reflects the usual requirement for transactions to be independent of each other. Assuming that this is the only form of restrictions in the set *Trans_Precedence*, the history value h will conform to the schema History_Invariant.

The schema Precede_Set produces the set *Precedes* based on the binding h to the component *History_Rec*. The set *Precedes* contains all the pairs of physical operations such that the first operation precedes the second in the sequence h.

The schema Conflicting_Set, based on the set *Precedes*, produces the set *Conflicting*. The set *Conflicting* contains all the members of *Precedes* that have conflicting operations.

The schema Conflicting_End_Points, based on the set *Conflicting*, produces the set *Conflicting_Points*. The set *Conflicting_Points* contains the transaction numbers of the transactions that are linked by a series of conflicting operations.

The schema Conflicting_Trans restricts all members of the set *Conflicting_Points* such that

$(n, n) \notin Conflicting\_Points$

It follows that the history, h, is a valid binding of the component *History_Rec* if it does not give rise to the set *Conflicting_Points* such that (n, n) is a member for any transaction number n.

Assume that i and j are transaction numbers that appear in h, such that $i \neq j$ and j contains an operation that conflicts with an operation in i where the operation in i occurs before the operation in j. hence

$(i, j) \in Conflicting\_Points$

For example, the value of h can have the form

$h = \ldots ri(xa) \ldots wj(xa) \ldots$

From the strict two phase locking mechanism of the implementation DBS_History, no operations in different transactions can conflict unless the first transaction has completed all its operations before the conflicting operation in the second transaction occurs. Hence, transaction i must have either a commit or an abort operation in history h before the occurrence of the conflicting operation in transaction j. For example, h can have the form

$h = \ldots ri(xa) \ldots ci \ldots wj(xa) \ldots$

This means that there can be no other operations in transaction i that occur after the conflicting operation in transaction j. Therefore, transaction i must complete before transaction j.

A cycle of the form

$(i, i) \in Conflicting\_Points$

will occur if the set *Conflicting_Points* is produced such that

$(i, j) \in Conflicting\_Points \wedge (j, i) \in Conflicting\_Points$

However, if

$(j, i) \in Conflicting\_Points$

then an operation in transaction j occurs before an operation in transaction i such that the operations conflict. Because of the strict two phase locking mechanism employed in the schema DBS_History, if

$(j, i) \in Conflicting\_Points$

then transaction j must have completed before the occurrence of the conflicting operation in transaction i. Therefore, transaction j must complete before transaction i. However, this contradicts the conclusion for the case

$(i, j) \in Conflicting\_Points$

Therefore, if

$(i, j) \in Conflicting\_Points$

then

$(j, i) \notin Conflicting\_Points$

It is also necessary to consider whether it is possible to form a path of conflicting transactions such that

$(i, j) \in Conflicting\_Points \wedge (j, k) \in Conflicting\_Points \wedge \ldots$

$\ldots (p, i) \in Conflicting\_Points$

This forms a chain of reasoning such that

i completes before j completes before k . . . completes before p completes before i

Since *completes before* is an obvious transitive and antisymmetric relation, the above

simplifies to

i completes before i

which is a contradiction and hence is not allowed by the two phase locking mechanism.

Therefore, any history value *h* produced by the schema DBS_History will give rise to the schema Conflicting_End_Points producing the set *Conflicting_Points* such that

$(n, n) \notin Conflicting\_Points$

The schema Conflicting_Op_Set produces the set *Conflicting_Op* by adding members to the set *Conflicting_Points* that result from conflicts between logical data objects. The protocol implemented by the schema Trans_Man_Req is a write all protocol. This has the effect making the distributed physical data objects perform as a single data object. That is, no data object is updated in isolation to the others and a write lock must be obtained from all the physical data objects. Therefore, the schemas Conflicting_Op_Set and Ordered_End_Points will not add any new members, hence

$Ordered\_Points = Conflicting\_Points$

and the condition

$(n, n) \notin Ordered\_Points$

in the schema One_Copy_Serialization is true, thereby indicating that the history value *h* is a valid binding of the component *History_Rec* and is therefore one copy serializable.

Note that the only restriction on the order operations in the implementation is that given by their order within transactions, hence the implementation does not meet the full flexibility represented by the specification. Should any additional precedence relation be required between transactions, as represented by the set *Trans_Precedence*, then the implementation must be changed.

5.7 Summary

This chapter contains a detailed study of an integrated implementation of the concurrency control aspects of a replicated database system. By integrated, it is meant that all the schemas are combined into a single, coherent model of the system that defines all the data necessary to identify each operation.

As revealed by the schemas contained in Chapter 3, it is useful to maintain a consistent style of writing schemas and in some cases this does entail extra variables to be existentially quantified. In most schemas, the preconditions are clearly separate from the postconditions, however for some schemas, such as Trans_Man_Read, the preconditions and postconditions are part of the same quantification because of the scope rules of the quantifiers. The predicates can be separated by repeating the quantified predicates; once to establish the precondition and a second time to define the postconditions. However, this is at the expense of further complicating the schema definitions.

The one copy serializability property is specified in the form of the schema Serializable_History that defines all possible histories that are one copy serializable, given the precedence relation determined by the set *Trans_Precedence*. Figure 5.1 indicates the construction of this schema from other schemas defined in Section 5.1 and provides a good summary of the relations between all the schemas defined in Section 5.1.

Interaction diagrams are employed in Sections 5.2 - 5.5 to emphasise the connection between sets schemas, such relations are not always apparent from the definitions in the Z notation.

The two proof sketches in Section 5.6 that verify the implementation are quite straightforward and elegant, although they lacked strict formality. However, as the specification and implementation are described precisely, the possibility of error is reduced compared to using natural languages, or semi formal languages. The completely formal proof (i.e. a proof demonstration), which although may be possible with computer assistance, is very difficult to complete manually. Also, the benefits of completely formal proofs have not been demonstrated to warrant the expenditure of effort required to perform them.

Proof sketches are capable of revealing hidden assumptions, for instance, the proof sketch in Section 5.6.2 revealed an underlying assumption of the implementation about the form of precedence relations expected between operations in the same transactions. The specification of the precedence relation is not as restrictive as that in the implementation and this mismatch may had not been apparent without the aid of proof sketches.

The proof obligations of this chapter are the same as those discharged in Chapter 3, see Section 3.8.2.